VŠB – Technická univerzita Ostrava
Fakulta Elektrotechniky a informatiky

DIPLOMOVÁ PRÁCE

2010 Radek Kaluža

VŠB – Technická univerzita Ostrava
Fakulta Elektrotechniky a informatiky
Katedra informatiky

Klient pro ERP systém na platformě Windows CE

ERP Client for Windows CE

2010 Radek Kaluža

Zadání

zadání z katedry..

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

Radek Kaluža

Poděkování

Chtěl bych poděkovat vedoucímu práce, panu Ing. Michalu Krumniklovi za jeho připomínky a podněty.

Abstrakt

Cílem této práce bylo vytvořit klientskou aplikaci pro mobilní zařízení. Její implementace je postavena nad existujícím ERP systémem vyvinutým firmou Xevos. Aplikace podporuje několik agend, které uživateli poskytují informace např. o firmách či skladových zásobách. Komunikace se serverovou částí probíhá pomocí webových služeb zabezpečených pomocí certifikátů.

Klíčová slova

.NET Compact Framework, WCF, webové služby, C#, Windows Mobile, certifikát, klient, formuláře, komponenty, ERP, Windows CE, MSSQL

Abstract

The aim of this work was to create a client application for mobile devices. Its implementation is built over an existing ERP system developed by Xevos. The application supports several agendas which provides information about companies, or inventory. Communication with the server is using Web services and their security is assured by certificates.

Keywords

.NET Compact Framework, WCF, web services, C#, Windows Mobile, certificate, klient, forms, components, ERP, Windows CE, MSSQL

Seznam použitých symbolů a zkratek

.NET	*.NET Framework* – běhové prostředí nad operačními systémy Microsoft Windows, poskytuje technologie a postupy pro vývoj aplikací na různá zařízení
.NET CF	*.NET Compact Framework* – „zmenšená“ verze .NET Frameworku pro mobilní zařízení
AJAX	*Asynchronous JavaScript and XML* – technologie určená pro vývoj webových aplikací, jejím účelem je u webových aplikací navodit dojem standardní formulářové aplikace, stránky nejsou znovu celé načítány, asynchronní požadavky využívají technologii XML
ERP	*Enterprise Resource Planning* – informační systém pro podporu běhu firmy, disponuje automatizovanými procesy, zahrnuje různé agendy – většinou jde o výrobu, logistiku, skladové hospodářství, účetnictví
GUI	*Graphical user interface* – grafické rozhraní, pomocí něhož uživatel komunikuje s aplikací
GUID	*Globally unique identifier* – speciální typ identifikátoru, který je používán v aplikacích za účelem jednoznačného značení, jedná se řetězec s 32 znaky jehož formát může vypadat následovně 44E99000-3AAA-6081-A2DD-06212B303123
IE	*Internet Explorer* – webový prohlížeč od firmy Microsoft
JMS	*Java Message Service* – aplikační rozhraní pro zasílání zpráv mezi dvěma klienty, spadá pod platformu JAVA Enterprise Edition
LINQ	*Language Integrated Query* – dotazovací jazyk spadající pod platformu .NET, představen s .NET Frameworkem 3.5, dává možnost dotazovat se nad různými zdroji dat, třídit je, hledat v nich
MSSQL	*Microsoft SQL Server* – databázová technologie od firmy Microsoft
MVC	*Model-View-Controller* – návrhový vzor / architektura, která děli aplikaci na tři vrstvy, zobrazovací, datovou a řídící logiku, jednotlivé části jsou na sobě nezávislé
RFID	*Radio Frequency Identification* – rádiový identifikátor sloužící k označení zboží, navazuje na systém čárových kódů, existují aktivní a pasivní identifikátory, mohou nést mnoho informací – záruka, trvanlivost apod.
SDK	*Software development kit* – soubor nástrojů pro vývojáře k usnadnění vývoje aplikací
SOA	*Service-oriented architecture* – architektura založena na službách a jejich konzumentech, zahrnuje principy tvorby SOA aplikací a klade na ně různé požadavky – znovupoužitelnost, modularitu, komponentní přístup
SQL	*Structured Query Language* – dotazovací jazyk používaný v relačních databázích
WCF	*Windows Communication Foundation* – aplikační rozhraní pro snadné budování aplikací orientovaných na služby (SOA)
WM 6.0	*Windows Mobile 6.0* – operační systém od firmy Microsoft pro mobilní zařízení, je vystavěn na jádře Windows CE

XML	*Extensible Markup Language* – značkovací jazyk nezávislý na platformě, určený k publikaci dokumentů, výměně dat mezi aplikacemi, službami apod., disponuje vysokou informační hodnotou a je pro obyčejného uživatele snadno čitelný
XMLDOM	*Document Object Model* – objektová reprezentace XML dokumentu, představuje taktéž aplikační rozhraní, určené pro přístup k obsahu XML dokumentu, jeho modifikaci

Obsah

Kapitola 1
Úvod

Doba se změnila - to co pro nás bylo před dvaceti lety jako vytažené z vědeckofantastického románu, je dnes skutečností. Technický pokrok nabral za poslední dvě desetiletí velkou rychlost a ani do budoucna to nevypadá, že by měl zpomalit či zastavit, spíše naopak.

Vzpomeňme si na dobu minulou, jak tehdy vypadaly televize, v jakých autech lidé jezdili, někteří movitější dokonce vlastnili mobilní telefon a jak to vypadá dnes. Není to ale jen technika, která pokročila, lékařství zaznamenalo mnoho úspěchů v léčení nemocí, čím dál více se mluví o genetice a klonování. Zábavní průmysl, finančnictví a další odvětví taktéž nezůstaly pozadu.

Podívejme se na odvětví informačních technologií. Každým rokem se zrychlují procesory, přidávají jádra, velikosti operačních pamětí v počítačích se počítají v gigabajtech, úložný prostor dokonce v terabajtech. Ruku v ruce jde s vylepšováním hardwaru i vývoj softwaru. Dnešní software je velmi složitý, sofistikovaný a robustní. Firmy vyvíjejí systémy všeho druhu, ať už je to systém pro řízení dopravy či navigační software do automobilu nebo jen aplikace pro podporu fungování firmy. Jedno však mají společné – pomáhají nám, usnadňují práci, život.

Právě softwaru, který slouží pro podporu procesů a usnadnění práce ve firmě je věnován tento text. Nebudeme se však zabývat konkrétně vývojem či problematikou těchto rozsáhlých systému, nýbrž se zaměříme na vývoj jakési nástavby v podobě klientské aplikace pro mobilní zařízení.

Dnešní mobilní zařízení, ať už PDA či „obyčejné“ telefony, jsou velmi vyspělé a dalo by se jim také říkat „příruční počítače“. Jejich výkon je totiž srovnatelný s počítači, které se začínaly objevovat počátkem devadesátých let. Taktéž jejich popularita nabrala na objemu a tyto přístroje jsou využívány všemi věkovými kategoriemi a to skrze všechna možná odvětví.

Čím dál větší množství firem, produkující software, začíná využívat výhod mobilních zařízení. Hlavní výhoda je zřejmá – mobilita. Ta dovoluje např. práci v terénu a eliminuje tak nutnost výsledky svojí práce zadávat do „statického“ počítače někde v kanceláři. Mobilních zařízení a jejich aplikací dnes využívá mnoho oborů a firem, mezi ně určitě patří elektrárenské společnosti, různí manažeři a obchodníci a dokonce i průvodčí ve vlacích.

Práce stručně popisuje technologie použité při vývoji řešení mobilního klienta, dozvíme se, jaká byla omezení pro použitou mobilní platformu a podíváme se i na samotnou implementaci. O ní se dočteme v kapitole 4, kde se také mimo jiné dozvíme, jaké komponenty byly použity, jak vypadá návrh formulářů a jakým způsobem bylo realizováno zabezpečení celého řešení.

Kapitola 2
Seznámení se s technologiemi

Kapitola seznamuje čtenáře s technologiemi použitými při samotné realizaci práce. Setkáme se zde s nástroji, které byly využity při vývoji jak na straně serveru, tak na straně mobilního klienta. Základními stavebními kameny jsou objektově orientovaný programovací jazyk C# verze 3.0, .NET Compact Framework, technologie WCF a databázový systém Microsoft SQL server.

2.1 Programovací jazyk C# 3.0

Jazyk C# firma Microsoft uvádí v roce 2002, kdy přišel na svět společně s .NET Frameworkem ve verzi 1.0. Od začátku je koncipován jako objektově orientovaný. Vychází a přebírá syntaxi jazyka C++, inspiraci taktéž hledá v konkurenčním objektově orientovaném jazyce Java.

Druhá verze tohoto jazyka je představena koncem roku 2005 a přináší novinky v podobě generických datových typů, částečných tříd, delegátů a dalších.

Koncem roku 2007 se objevuje nová verze s číselným označením 3.0 a vychází společně s .NET frameworkem 3.5. Opět přináší velké množství novinek, mezi nimi např. LINQ, což je integrovaný dotazovací jazyk, umožňující programátorovi dotazovat se na data, hledat v nich, třídit. Tento jazyk lze použít nad objekty, databází a XML soubory. Při vývoji tento jazyk nebyl použit a proto se jím nebudeme dále zabývat. Dalšími novinkami jazyka C# 3.0 jsou inicializátory objektů a kolekcí, rozšiřující metody a klíčové slovo *var*.

Mezi vlastnosti tohoto jazyka také patří např. automatický sběr neplatných objektů a uvolňování paměti, tzv. garbage collecting, jednoduchá dědičnost (třída může dědit jen z jedné třídy), zapouzdření atributů třídy pomocí tzv. *Property*, kdy navenek vystupuje atribut jako veřejný, ale přístup k jeho hodnotě je zapouzdřen definovatelnými *get* a *set* bloky. Což usnadňuje a zpřehledňuje kód a práci. (v Javě je tento přístup realizován pomocí metod *GetProměnná* a *SetProměnná*)

C# je tedy vysokoúrovňový objektově orientovaný programovací jazyk využitelný k tvorbě webových aplikací, webových služeb, formulářových programů a aplikací pro mobilní zařízení. Donedávna byl ovšem vývoj těchto aplikací omezen jen na operační systém Windows, dnes je možné vyvíjet software v jazyce C# i na jiné platformy.

Právě díky projektům jako je Mono[4], máme šanci vyvíjet programy, které budou fungovat i na operačních systémech Linux a Mac OS. Díky tomu, že je Mono koncipován jako multiplatformní (cross-platform), dokáže běžet i na zařízeních jako je Sony Playstation, Nintedno Wii nebo Apple iPhone. Kromě podpory různých architektur (x86, x86-64, ARM) má otevřený kód a tudíž není problém jej modifikovat.

2.2 Windows Mobile 6.0

Windows Mobile 6.0 (WM) je operační systém vyvinutý firmou Microsoft běžící na jádře Windows CE verze 5.2. Pro lepší pochopení se vraťme zpět do roku 1996, kdy vznikla první verze Windows CE.

Windows CE, taktéž oficiálně nazývaný Windows Embedded Compact či Windows Embedded CE je operační systém určený pro zařízení s omezenou operační pamětí a úložným prostorem. První verze vzniká roku 1996, další se pak objevují každé dva roky, přičemž zatím poslední verze 6.0 byla představena v září 2006. Během těchto let systém prošel mnoha změnami a stal se z něj komponentově orientovaný systém pracující v reálném čase, který už nebyl zaměřen čistě jen na kapacitně omezená zařízení.

Paralelně s vývojem Windows CE dochází také k vývoji nových operačních systémů určených pro tzv. „chytré“ telefony (smartphones). Prvním z nich nese označení Pocket PC 2000. Jak označení napovídá, tento systém byl představen v roce 2000. Obsahoval několik vestavěných aplikací, jejichž plnohodnotné ekvivalenty jsou vám určitě známy. Jedná se o Pocket Word, Pocket Excel, Pocket Internet Explorer, Windows media player a další.

O tři roky později od první verze těchto nových systému se mění konvence pojmenování a objevuje se nový název – Windows Mobile. WM 2003, WM 2003SE, WM 5.0, tyto systémy předcházely na začátku zmiňovanému WM 6.0, pro který bylo řešení této práce implementováno.

WM 6.0 je, jak jsem se zmínil v úvodu kapitoly, postaven na jádře Windows CE 5.2 z roku 2004. Toto jádro již podporovalo např. technologii Bluetooth, nové architektury procesorů, pokročilou 3D grafiku a mnoho dalšího. WM 6.0 přidává spoustu nové funkcionality a podporu mnoha dalších technologií např. AJAX, JavaScript a XMLDOM v prohlížeči IE Mobile, .NET Compact Framework v2 SP2 o jehož pozdější verzi bude řeč v následující kapitole.

2.3 .NET Compact Framework

.NET CF je rámec (Framework) spadající pod obecně velmi známý .NET Framework. Použití je zaměřeno na mobilní přístroje, jako jsou PDA, mobilní telefony a další podobná zařízení.

Framework byl navržen pro běh na platformě Windows CE, o které jsme si povídali v kapitole 2.2, to tedy znamená, že je samozřejmě použitelný i pro operační systémy s označením Windows Mobile. Dalo by se říct, že .NET CF je kopií „velkého“ .NET Frameworku, tedy že využívá stejné knihovny pro svou práci. To je z části pravda, ale díky kapacitním omezením, ať už výkonnostním či paměťovým, došlo k omezení (některé jmenné prostory nejsou k dispozici, omezení u WCF viz. kapitola 2.3.1) jeho funkcionality a v některých případech i implementaci nových knihoven, používajících specifika mobilních zařízení. Pro lepší představu jmenujme např. třídy pro práci s GPS, vstupním panelem nebo sériovými porty.

V roce 2002 vychází verze 1.0 a během několika let se vyvíjí až do verze 3.5, která byla uvolněna v lednu roku 2008. V tuto chvíli je ve vývoji verze 3.7, jejíž datum vydání ještě nebylo specifikováno.

Řešení této práce je implementováno nad verzí 3.5 a pro správnou funkčnost je její instalace na mobilním zařízení vyžadována. U starších modelů mobilních přístrojů je nutné tuto verzi doinstalovat, u přístrojů s operačním systémem Windows Mobile 6.5 jsou již tyto knihovny předinstalovány.

2.4 Windows Communication Foundation (WCF)

Při hledání vhodné technologie, která by mi pomohla ulehčit přenos dat mezi mobilním klientem a serverem jsem narazil právě na technologii WCF. Zásadním způsobem zjednodušila komunikaci, ovšem nic není zadarmo a i WCF má své nevýhody a tím více se bylo potřeba zamyslet nad jejím použitím na mobilní platformě. O omezeních si povíme v podkapitole 2.4.5. Nyní přejděme k výkladu základů.

WCF je Framework pomocí něhož můžeme vytvářet aplikace orientované na webové služby, též označované jako SOA aplikace. Ty jsou založeny na komunikaci mezi službou a klientem, případně mezi dvěma a více službami. Framework byl navržen tak, aby co nejvíce usnadnil práci programátorům, nebyl závislý na jiné technologii, přenos dat byl spolehlivý, rychlý a poskytoval jednoduché nastavení zabezpečení.

Pro práci s WCF je důležité znát a vědět o těchto pojmech:

- Služba
- Koncový bod
- Vázání
- Zpráva
- Proxy

2.4.1 Služba

Služba je něco, co vy poskytujete navenek, např. v internetu a dovolujete tím tak komunikovat klientům s vaší službou. (případně jiné služby) V aplikaci můžete mít jednu službu, ale nic vám nebrání mít služeb více, pro různé účely.

Co všechno musí mít taková služba definováno? V prvé řadě je to třída, jejíž implementace se bude starat o obsluhu příchozích požadavků. Dalším krokem je definice koncových bodů. Těch může existovat i více než jeden, v případě mého řešení jsem ale tuto možnost nevyužil.

```
<%@ServiceHost
Language="C#" Debug="true"
Service="MobieERPServices.MobileERP.Implementation.Managers.OsobaManagerImpl"
CodeBehind="MobieERPServices.MobileERP.Implementation.Managers.OsobaManagerImp
l.cs" %>
```

Obrázek 2.1: Definice služby v souboru OsobaManagerService.svc

Obrázek 2.1 ukazuje, jakým způsobem je definována služba, která se stará o záležitosti okolo osob – zasílání seznamů, získávání konkrétních záznamů atp. Do vlastnosti *Service* a *CodeBehind* je přiřazena třída implementující tyto obslužné vlastnosti, při vývoji jsem také používal tzv. *Debug* mód, vypisující podrobné chybové hlášení.

Co musí splňovat implementace třídy, kterou chceme vystavit jako službu? Jedinou podmínkou je, že musí implementovat rozhraní, které bude označeno speciálními atributy. Mezi tyto atributy patří:

- ServiceContract
- OperationContract
- DataContract
- DataMember

První dva atributy, *ServiceContract* a *OperationContract* se týkají rozhraní pro službu a zbylé dva uvedené *DataContract* a *DataMember* se používají u tříd se kterými rozhraní pracuje.

Atributem *ServiceContract* značíme rozhraní, které, jak jsem se již zmínil, musí implementovat třída, jejíž název je určen v souboru představující službu. Označení je velmi jednoduché, jak je vidět na obrázku 2.2. Taktéž je vidět jakým způsobem se používá atribut *OperationContract,* určený k označení metod rozhraní. Tyto metody pak bude služba „poskytovat“ navenek a bude možné je volat za pomocí klientů.

```
[ServiceContract]
public interface IOsobaManager {

[OperationContract]
List<OsobaGridObject> GetListMobile(int from, int to);
...
}
```

Obrázek 2.2: Rozhraní služby pro obsluhu osob

Jistě jste si všimli, že metoda *GetListMobile* vrací generický list objektů *OsobaGridObject*. Aby toho byla služba schopna a všechno fungovalo, je potřeba „označit“ tuto třídu speciálními atributy tak, aby o ní WCF Framework věděl a uměl ji použít. Označením atributů říkáme, které z nich se mají zasílat ve zprávách na „druhou“ stranu.

```
[DataContract]
public class OsobaGridObject
{
 [DataMember]
 public int Id { get; set; }
 [DataMember]
 public string Jmeno_prijmeni { get; set; }
 [DataMember]
 public string Kontakt { get; set; }
 [DataMember]
 public string Pobocka { get; set; }
}
```

Obrázek 2.3: Třída využívána službou pracující nad agendou osob

Atributem *DataContract* označíme všechny třídy o kterých víme, že budou přenášeny ze služby na klienta. Pomocí *DataMember* atributu si můžeme „zvolit“ které z vlastností (property) té které třídy chceme přenášet mezi službou a klientem. Ukázka kódu na obrázku 2.3 je jednoduchá, ale pokud bychom měli třídu obsahující další třídy, vlastnosti z jiných tříd (např. vlastnost Pobocka z obrázku 2.2 by nebyla datového typu *string*, ale typu *Pobocka*), tak je nutné tyto používané třídy taktéž označit atributy *DataContract*, případně *DataMember*.

2.4.2 Koncový bod

V předchozí podkapitole jsem se zmínil o tom, že je nutné specifikovat jeden či více koncových bodů pro jednotlivé služby. Koncový bod je v podstatě adresa, na které bude naše služba k nalezení. Při jeho definování se obvykle specifikuje vázání a konkrétní nastavení vázání. Co je to vázání a jak je definováno uvidíme v následující podkapitole.

```
<endpoint
address="http://mobile.biggie.xevos.cz/OsobaManagerService.svc"
binding="basicHttpBinding" bindingConfiguration="ManagerBinding"
contract=
"MobieERPServices.MobileERP.Implementation.Interfaces.IOsobaManager">
</endpoint>
```

Obrázek 2.4: Ukázka definice koncového bodu se specifikací vázání

Za povšimnutí určitě stojí atribut *contract*, jemuž je přiřazeno rozhraní, o kterém byla řeč v podkapitole 2.4.1.

2.4.3 Vazba

Vazba, též v originále *binding*, slouží k definování detailů jak pro službu, tak pro klienty. Jeho pomocí definujeme detaily o transportu zpráv, kódování a zabezpečení. WCF Framework nabízí velké množství vazeb, jejichž použití může záviset na řešeném problému. Vyjmenujme si několik základních:

- BasicHttpBinding
- WSHttpBinding
- NetTcpBinding

Každá z vyjmenovaných vazeb má různé možnosti zabezpečení, respektive ve výchozím nastavení využívá jiný typ. Příkladem je *BasicHttpBinding*, jež ve výchozím nastavení nevyužívá žádného zabezpečení, na rozdíl od *WSHttpBinding*, které definuje zabezpečení na úrovni zpráv. Samozřejmě, že je možné si nastavení nakonfigurovat podle

libosti. Rozdíly nejsou však jen v zabezpečení, ale také např. v podpoře transakčního přenosu, podpoře duplexní komunikace atd. Téma vazeb je obsáhlejší a více se o něm dozvíte v [1].

```
<basicHttpBinding>
 <binding name="ManagerBinding" maxBufferSize="655360"
  maxReceivedMessageSize="655360">
  <readerQuotas maxDepth="64" maxStringContentLength="2147483647"
  maxArrayLength="2147483647" maxBytesPerRead="4096"
  maxNameTableCharCount="16384" />
 </binding>
</basicHttpBinding>
```

Obrázek 2.5: Ukázka konfigurace vazby *BasicHttpBinding*

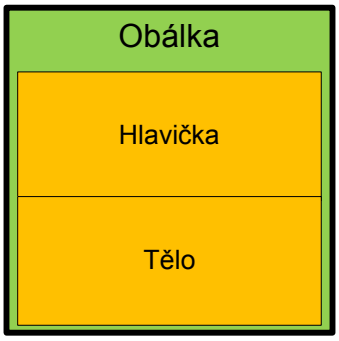
V rychlosti si ještě popišme některé atributy z obrázku 2.5. Důležité je nastavení atributu *maxRecievedMessageSize* jež definuje jak velké zprávy bude služba posílat a klient přijímat. Problém totiž může nastat ve chvíli, kdy chcete zasílat zprávy velikosti větší než je nastavená výchozí (65536 bajtů).

Element *readerQuotas* slouží k nastavení omezení SOAP zpráv. Hloubku zanoření elementů ve zprávě určíme celočíselnou hodnotou atributu *maxDepth*. Zajímavým atributem je i *maxStringContentLength,* díky němuž můžeme definovat, jak velký obsah se může nacházet v elementu zprávy. To se hodí zejména v případě, pokud posíláte dlouhé řetězce.

2.4.4 Zpráva

V kapitole 2.4.3 jsme si řekli, že existují nějaké vazby. V našem řešení byla použita vazba *BasicHttpBinding* z důvodu omezení, o kterých se dozvíme v kapitole 2.4.6. Jednou z vlastností této vazby je, že využívá HTTP protokol k přenosu zpráv. Tyto zprávy jsou postaveny na protokolu SOAP, který je založen na XML.

Jak taková SOAP zpráva vypadá, naznačuje obrázek 2.6. Hlavním „kontejnerem“ je obálka, ve které se nachází dva další elementy – hlavička a tělo zprávy.



Obrázek 2.6: Struktura SOAP zprávy

SOAP[2] protokol existuje ve dvou verzích, 1.1 a 1.2. Vazba *BasicHttpBinding* využívá jeho starší verzi. Protokol má několik výhod, ale má i své nevýhody. Výhodou je možnost použití i jiného transportního protokolu, např. JMS[3]. Nevýhodou je obsáhlost zprávy a to díky XML struktuře. Mnohdy se stává, že samotná přenášená data ve zprávě jsou menší než XML kterým jsou „obalena“.

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
 <s:Header>
  <Action s:mustUnderstand="1"
   xmlns="http://schemas.microsoft.com/ws/2005/05/addressing/none">
   http://tempuri.org/IOsobaManager/Count
  </Action>
 </s:Header>
 <s:Body>
   <Count xmlns="http://tempuri.org/" />
 </s:Body>
</s:Envelope>
```

Obrázek 2.7: Zpráva požadavku o zjištění počtu osob

Odpověď na požadavek vypadá následovně (obrázek 2.8), všimněte si, že získaná data (počet osob) jsou několikrát menší než samotné XML. S přibývající složitostí zpráv, přibývá i popisné XML a ve výsledku může mít zpráva desítky, stovky kilobajtů. To je v mnoha případech problém, jehož řešením může být např. použití komprese či jiné techniky.

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
 <s:Header>
  <ActivityId CorrelationId="34bafbc6-03c9-409b-8409-5c3a63456381"
xmlns="http://schemas.microsoft.com/2004/09/ServiceModel/Diagnostics">
      13655ec3-9095-4e8c-af4e-17d69badf16d
  </ActivityId>
 </s:Header>
 <s:Body>
  <CountResponse xmlns="http://tempuri.org/">
   <CountResult>21</CountResult>
  </CountResponse>
  </s:Body>
</s:Envelope>
```

Obrázek 2.8: Odpověď na požadavek o zjištění počtu osob

2.4.5 Proxy

Proxy - do českého jazyka přeloženo jako *náhradník, zastoupení*, je třída používaná na straně klienta za účelem komunikace se službou. Pomocí instance této třídy dojde k navázání spojení a následnému přenosu informací. Ten je realizován díky tomu, že proxy třída obsahuje metody dané služby.

Jak takovou proxy třídu získáme? Existuje několik způsobů jak ji vytvořit. Jedním z nich je tzv. „ruční" implementace. Nic vám totiž nebrání si tuto třídu napsat od úplného začátku, nevýhodou, respektive požadavkem je rozsáhlejší znalost WCF Framework.

Druhá možnost je mnohem jednodušší a spočívá v použití nástrojů, které byly uvolněny společně s. NET Frameworkem. Ty ulehčují práci vývojářům tím, že generují proxy třídy na základě předané adresy běžící služby. Jejich funkce je založena na zpracování vystavených metadat dané služby, které je nástroj schopen číst a na jejich základě generovat třídy.

Nástroje, o kterých je řeč, jsou konzolové aplikace s relativně jednoduchým ovládáním a možností konfigurace pomocí předaných parametrů. Existuje verze používaná v rámci vývoje aplikací na operačních systémech Windows a nazývá se *ServiceModel Metadata Utility Tool.* Standartně je k nalezení v adresáři *<váš disk>\Program Files\Microsoft SDKs\Windows\v6.0\Bin*. Pokud byste však v tomto adresáři hledali soubor s tímto dlouhým názvem, nejspíš byste neuspěli – nástroj nese název *svcutil.exe*.

Obdobná situace je i pro mobilní vývoj na platformě Windows CE, kde existuje identický nástroj pro automatické generování proxy tříd. Nazývá se *The .NET Compact Framework ServiceModel Metadata Tool* a naleznete jej v instalačním adresáři .NET Compact Frameworku, kterému je věnována kapitola 2.3.

Použití nástroje pro mobilní platformu je snadné a může vypadat např. takto:

NetCFSvcUtil.exe http://mobile.biggie.xevos.cz/FirmaManagerService.svc /out:FirmaManagerImpl.cs

Prvním parametrem je adresa webové služby, následuje parametr */out*, který určuje název souboru proxy třídy. V nápovědě k aplikaci existuje podrobný popis jednotlivých parametrů, z nichž si ještě dovolím jmenovat */language:<language>* kde *<language>* může být zastoupen hodnotami *c#, cs, csharp, vb, visualbasic, c++, cpp*, pokud však není parametr specifikován, výchozí hodnotou je *csharp*.

2.4.6 Omezené možnosti v .NET Compact Frameworku

Po představení .NET Compact Frameworku z kapitoly 2.3, jsme se vrhli na technologii WCF. Během popisu základních pojmů WCF Frameworku jsem se zmínil o jistých omezeních jeho použití na mobilní platformě. O jaká omezení se jedná, si krátce povíme v této kapitole.

Stejně jako je .NET Compact Framework omezenou podmnožinou „velkého" .NET Frameworku, tak existuje i podmnožina WCF Frameworku, použitelná na mobilní platformě. Jinými slovy řečeno, ne vše co WCF Framework nabízí lze použít v prostředí pro mobilní zařízení.

Co všechno zasáhla zmiňovaná omezení? Došlo k redukci podporovaných vazeb (bindings – viz. podkapitola 2.4.3), razantním restrikcím v zabezpečení – dvojice uživatelské jméno / heslo není podporována a jediným podporovaným šifrovacím algoritmem je *Basic256Rsa15*. Dobrou zprávou je podpora certifikátů, ale o nich bude řeč až v kapitole 4.8. Dalšími omezeními jsou – nemožnost šifrovat hlavičky zpráv (viz obrázek 2.6) a nespolehlivé doručování zpráv. Obzvlášť poslední omezení může vyvolat jisté obavy. Přece jen, ani dnes není připojení na mobilních zařízeních úplně stabilní. Seznam nepodporovaných částí je obsáhlejší a výše jmenované jsou jen malým fragmentem z celkového výčtu. Pro detailnější studium doporučuji [5] a [6].

2.4 Microsoft SQL Server

Zatímco je .NET Compact Framework (viz. kapitola 2.3) srdcem klientské aplikace, přesně na opačné straně, tedy serverové, je to MSSQL Server.

Microsoft SQL Server je, jak název napovídá, systém od firmy Microsoft využívající technologii SQL. Jazyk strukturovaných dotazů, zkráceně SQL, se používá k manipulaci s daty v relačních databázích. Jeho historie sahá až do 70. let minulého století, kdy se jeho vývojem začala zabývat firma IBM. Během několika let se k IBM přidalo několik dalších firem a postupem času se z původního jazyka SEQUEL (Structured English Query Language) stal jazyk SQL, tak jak ho známe dnes. Vraťme se ještě k pojmu relační databáze.

Pojem relační databáze byl poprvé definován panem E. F. Coddem v roce 1969. Jeho představa byla založena na matematických množinách a predikátové logice. Databázová relace zavádí pojem schéma relace. Ten zahrnuje název relace, počet sloupců a obor hodnot pro daný sloupec, taktéž nazývaný doména. V relačních databázích je schématem definice struktury tabulky. Ty jsou základem každé relační databáze. Jednotlivé tabulky mají stanoven název, definovány sloupce a jejich datové typy. V tabulkách se pak může nacházet n záznamů (řádků), těm se říká data.

MSSQL Server je systémem relační databáze, jehož počátky vzniku sahají do roku 1989, kdy se několik významných firem (Microsoft, Sybase, Ashton-Tate) na poli vývoje databází spojilo a vytvořilo první verzi běžící na operačním systému OS/2 – SQL Server 1.0.

Vývoj databázových systému pokračoval a v roce 1993 přichází Microsoft s SQL Serverem verze 4.21 pro nové operační systémy Windows NT. Během nástupu těchto operačních systému došlo k rozdělení dřívějšího spojení firem Microsoft a Sybase. Firmy měly na věc rozdílný pohled (jiné postupy) a nakonec došlo k tomu, že Sybase oddělila a přejmenovala svůj produkt na Adaptive Server Enterprise, jež se vydal svou vlastní cestou. Microsoft dále pokračoval ve vývoji SQL Serveru a v roce 2000 vydal verzi SQL Server 2000, která se stala populární a úspěšnou. Dnes, o deset let později, je možné vyvíjet aplikace založené na databázovém systému MSSQL Server 2008 R2, jež disponuje nespočetným množstvím různých funkcí. Jmenujme např. řízení souběžnosti[7], transakce, replikace dat, vyhledávání (full-textové) a další.

Za zmínku určitě stojí informace o existenci různých verzí či edicí, chcete-li. Jednotlivé edice jsou zaměřeny na různé potřeby zákazníka. Ať je to *Enterprise* edice, vhodná pro vývoj velkých řešení, či levnější *Web* edice, poskytující mnoho funkcionality za nízkou cenu nebo jen *Express* edice, která je zdarma a je vhodná např. pro studijní účely, malé aplikace.

Kromě jmenovaných edicí existuje také verze použitelná v mobilních zařízeních, nazývána *SQL Server Compact* a v minulosti známá pod označením *SQL Server for Windows CE*. Zmínil jsem slovo „použitelná“, což znamená, že již není primárně určena mobilním zařízením, může být totiž využita i na stolních počítačích.

Instalace této „odlehčené“ verze zabírá necelé 2 megabajty a běží v rámci aplikace, ve které ji používáte. To znamená, že nemůže běžet jako služba. Celá ta „magie“ se pak ukrývá v souboru s příponou *.sdf*. Práce s touto databází je identická jako s „velkou“ verzí. K dispozici je aplikační rozhraní ADO.NET poskytující přístup k datům, dále podpora LINQu a Entity Frameworku.

Kapitola 3
Aktuální situace

V této kapitole se podíváme na situaci kolem celého řešení. Nad jakým systémem je mobilní klient vystavěn, v jaké se momentálně nachází fázi, řekneme si krátce něco o jeho architektuře. Kapitola, která bude následovat je zaměřena na databázi a záležitosti kolem ní.

3.1 Systém

Jak název práce napovídá, systém nad kterým mobilní klient pracuje, nese zkratku ERP. Co to ovšem znamená?

ERP je zkratka pro anglický výraz *Enterprise resource planning* – podnikové plánování zdrojů. ERP systémy jsou velké softwarové balíky zahrnující velké množství modulů, které integrují a automatizují procesy ve firmě. Jinými slovy zefektivňují, zrychlují práci a centralizují data. Jednotlivé moduly bývají obvykle provázány, což usnadňuje operace skrze různé agendy. Standardně do množiny těchto modulů spadá výroba, fakturace, objednávky, logistika a účetnictví. Samozřejmě tento výčet nemusí být konečný a záleží na té které firmě, jaké další moduly naimplementuje do svého systému.

Svůj vlastní systém vyvíjí i firma *Xevos*. Jeho prozatímní název je *Biggie*. S tímto označením jsme se již setkali na některých obrázcích a určitě se objeví i v kapitole 4. *Biggie* je ERP systém vyvíjený na platformě .NET. Většina velkých ERP systémů je navržena jako formulářové aplikace. *Biggie* se však vydal opačným směrem a je koncipován jako aplikace webová. Určitě nemusím zmiňovat hlavní výhodu webové aplikace – přístupnost odkudkoliv z internetu. Ale taktéž jste si určitě vědomi toho, že to s sebou přináší i větší množství nevýhod – práce nad bezstavovým HTTP protokolem, zabezpečení.

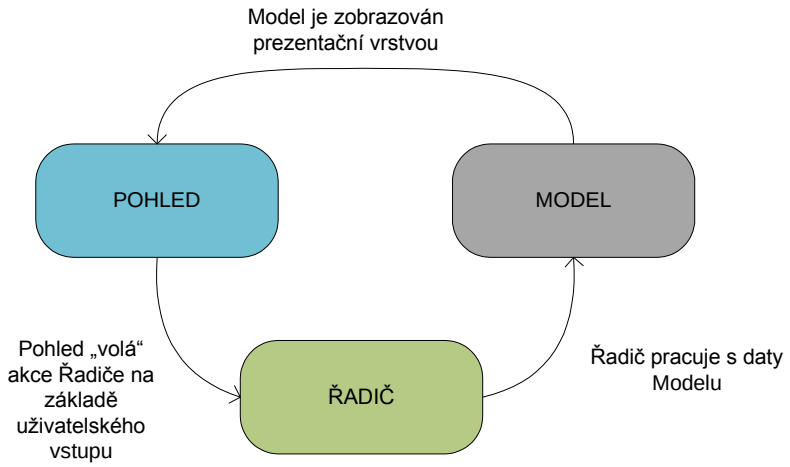
Systém je postaven na technologii ASP.NET a doplňují jej další jako je JavaScript, AJAX, webové služby. Pod tím vším se ukrývá databázový systém Microsoft SQL Server 2008 (viz. kapitola 2.4) představující úložiště celého systému.

Biggie je implementován podle vzoru / architektury MVC[12] (také nazývaným Model-2), tedy model-pohled-řadič. Jedná se o tří vrstvou architekturu, jejímž cílem je izolovat jednotlivé části aplikace do samostatných komponent. Díky jejich nezávislosti je možné jednotlivé komponenty nahrazovat aniž by muselo dojít k přepisování kódu ostatních částí. Tuto vlastnost jsem samozřejmě využil, jakým způsobem se dozvíte v kapitole 4.1.

Pojďme se podívat na to, co se v oněch „částech“ aplikace skrývá. První ze jmenovaných vrstev je *Model*. Většinou se jedná o tzv. „datovou“ vrstvu, jež má za úkol zprostředkovávat přístup do databáze. (případně mapovat objekty na tabulky atp.) Díky jasně definovanému *Modelu* (např. pomocí rozhraní) je pak možné změnit např. databázi, aniž by to jakkoliv postihlo vyvíjenou aplikaci. (nebo rovnou celou implementaci *Modelu*)

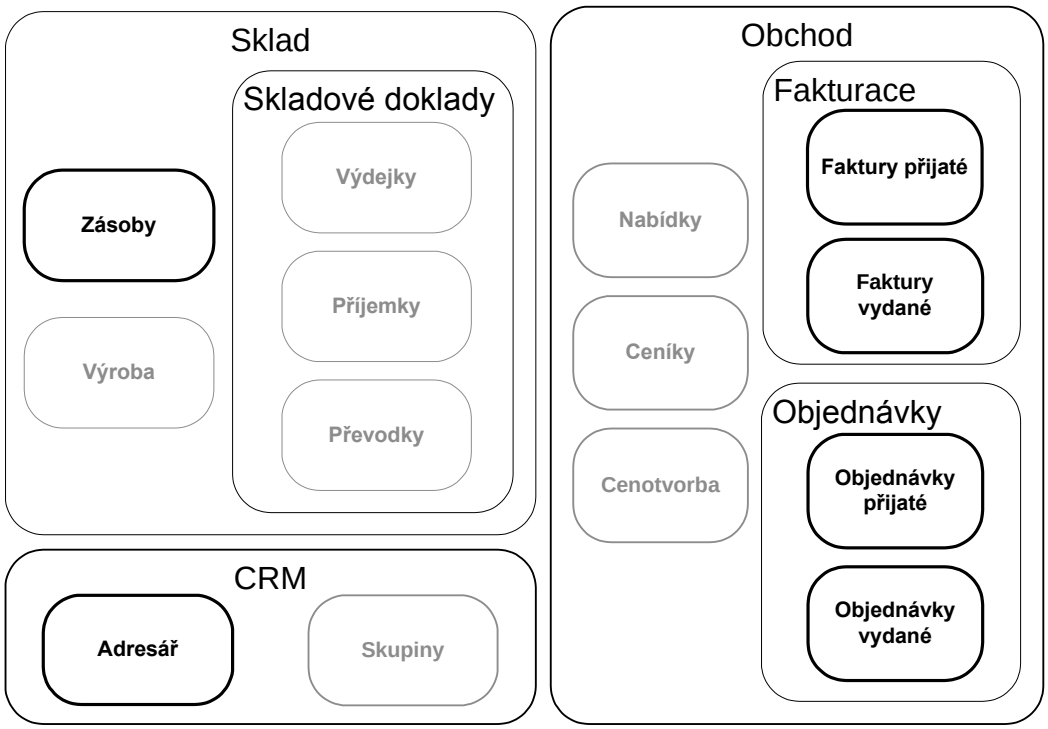
Další vrstvou je *Řadič* (Controller), který bývá v aplikacích implementován jako byznys logika systému. Tedy něco, co využívá vrstvu pod ním (Model) a poskytuje operace navenek, využitelné ve vyšších vrstvách.

Poslední vrstvou je Pohled (View). Dalo by se říct, že úkolem Pohledu je zobrazit nějaká data, získaná z Řadiče. Abychom si to lépe představili, uvedu příklad. Mějme ERP systém implementující tří vrstvou architekturu. Tento systém běží na technologii ASP.NET, tedy v prostředí webového prohlížeče. Právě technologie ASP.NET zde sehrává roli Pohledu. Pokud by si zadavatel řešení přál mít svůj systém implementovaný např. ve Flashi či Silverlightu, kód Modelu a Řadiče by se nezměnil a došlo by jen k „výměně“ zobrazovací (Pohled) vrstvy. A právě v tomto přístupu spočívá největší výhoda této architektury – v nezávislosti jednotlivých vrstev.



Obrázek 2.10: Architektura Model-Pohled-Řadič

V úvodu kapitoly jsme si řekli, jaké moduly bývají ERP systémy podporovány. Následující obrázek nám představí ERP systém Biggie z pohledu jím podporovaných modulů. Z důvodu přehlednosti a velikosti není diagram kompletní a výčet modulů je tak omezen na ty, v nichž se nacházejí agendy podporované klientem.



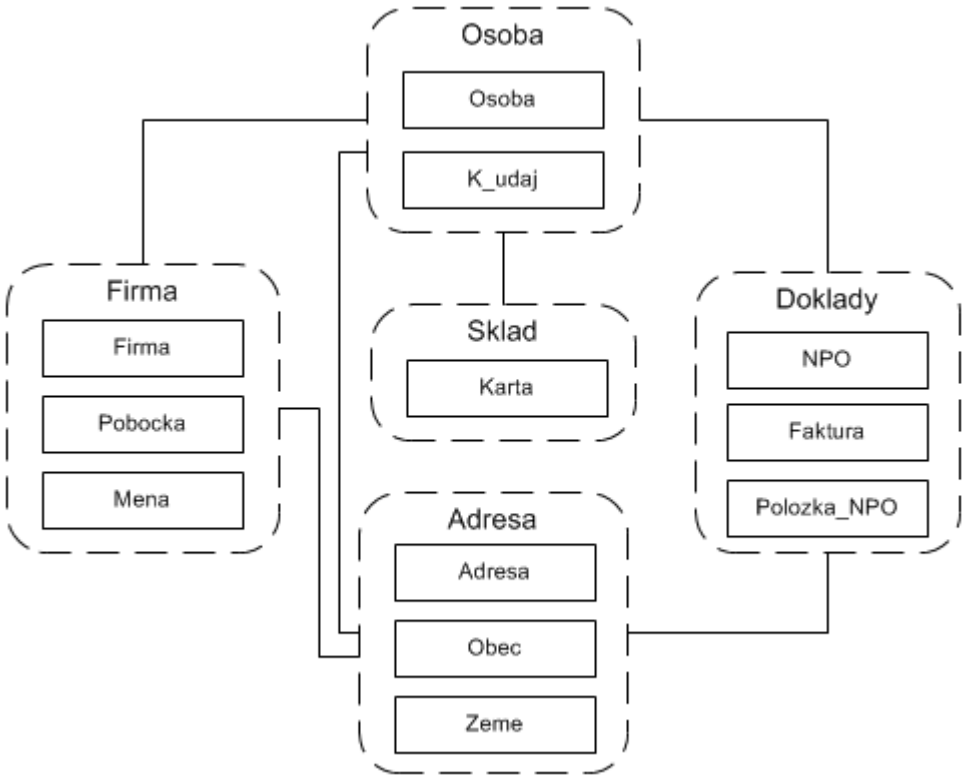
Obrázek 2.11: Moduly a agendy ERP systému Biggie

Tučně vyznačené agendy jsou ty, které podporuje mobilní klient, ostatní jsou k dispozici v rámci ERP systému. Na obrázku není vyznačena agenda *Identifikace*, která je podporována jen ze strany klienta.

3.2 Databáze

Databáze systému *Biggie* by se dala svou velikostí zařadit do kategorie středně velkých databází. Obsahuje přibližně stovku tabulek, větší množství uložených procedur a desítky pohledů.

Při řešení mobilního klienta nebylo zapotřebí do databáze zasahovat. Nezbytným krokem ovšem bylo vytvoření její kopie z důvodu vývoje a modifikací ze strany vývojářů, kteří na ní pracovali. Bez této kopie by byl vývoj mobilního klienta ztížen neustálými změnami, ať už tabulek či atributů.



Obrázek 2.12: Zjednodušený pohled na databázi ERP systému Biggie

Díky rozsáhlosti struktury databáze ERP systému, jsem na obrázku 2.12 vybral několik stěžejních tabulek. Ty byly zařazeny do logických celků vyznačených přerušovanou čarou. Na obrázku nejsou znázorněny vztahy mezi tabulkami z důvodu zachování přehlednosti.

Účel většiny tabulek je jasný z jejich názvu. Tabulka, která může vznést otázky je ta s názvem *NPO*. Je určena pro doklady nabídek, poptávek a objednávek. Jednotlivé položky dokladů – faktur a objednávek jsou ukládány do tabulky Polozka_NPO. Jak je vidět z obrázku, všechny logické celky jsou spojeny s osobou. Důvodem je možnost sledování vkládání jednotlivých záznamů do systému.

Kapitola 4
Řešení

Kapitola postupně odhaluje vnitřní strukturu samotného řešení mobilního klienta. Úvodní kapitoly nám poskytnou náhled na architekturu serverového a klientského řešení. Na úvod naváží kapitoly, jejichž cílem je objasnit opatření, která byla provedena z důvodu omezených prostředků mobilní platformy. Dále se podíváme na použité komponenty, které byly implementovány a použity ve formulářích. O těch bude řeč v kapitole 4.6. Závěr je věnován popisu jednoduchého odpojeného (offline) módu a zabezpečení.

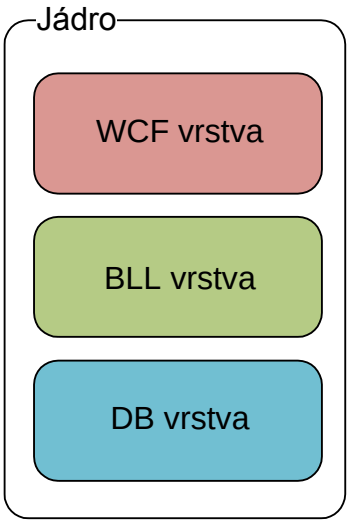
4.1 Architektura

Kód na straně serveru i klienta je rozvrstven, částečně díky původní architektuře ERP systému. Jak vypadají jednotlivé vrstvy a co obsahují, ukáží obrázky v následujících kapitolách.

4.1.1 Server

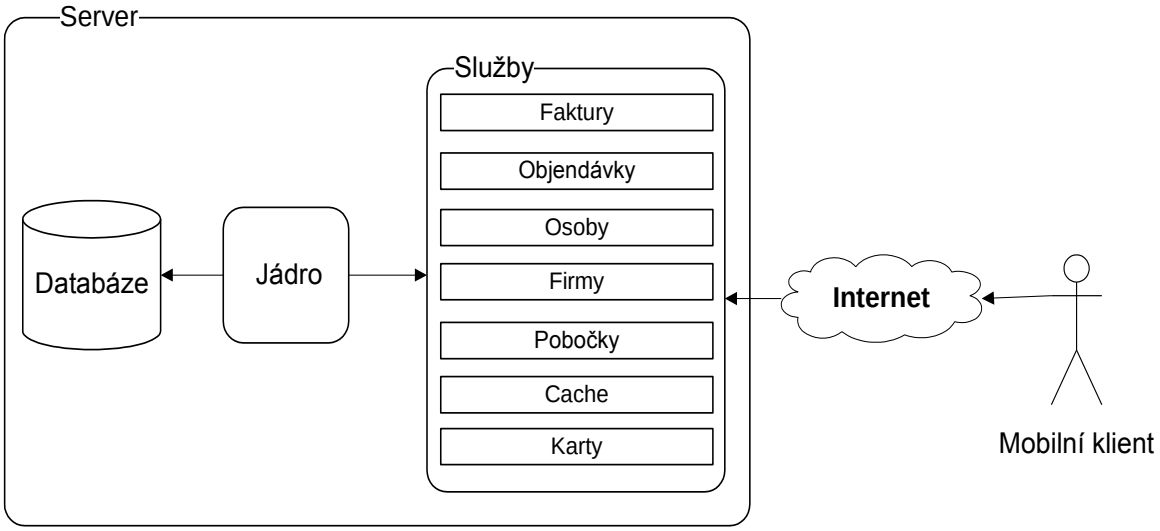
V kapitole 3.1 jsme se dozvěděli co je to ERP systém, jakým způsobem je implementován a že byl navržen jako vícevrstvý. Díky objasnění návrhového vzoru Model-Pohled-Řadič z kapitoly 3.1 je nám jasné, jakou má tato architektura výhodu. Nezávislost vrstev mi dovolila využít již existující kód vrstvy Modelu a Řadiče, který byl následně modifikován pro potřeby WCF Frameworku, coby zprostředkovatele komunikace mezi serverem a klientem.

Modifikovaný kód bylo ještě nutné obohatit o nové funkce, které využívaly „omezené“ objekty. (viz. kapitola 4.4) Ve „finále“ vznikla nová vrstva, vybudovaná nad dvěma existujícími. Jakou má kód, po modifikacích, na straně serveru architekturu ukazuje obrázek 4.1.



Obrázek 4.1: Architektura jádra serverové strany aplikace

Jelikož uživatelské rozhraní mobilního klienta není přesnou kopií rozhraní ERP systému, množina podporovaných agend byla zredukována a díky tomu byl využit jen zlomek tříd implementující byznys logiku systému. Při hledání vhodného řešení, jakým způsobem vyřešit „obalení“ těchto tříd specifickými WCF atributy, jsem došel k závěru, že nejlepší bude vytvořit poměr 1:1. Namísto implementace jediné „velké“ služby, která by zakrývala všechny podporované agendy, jsem tedy zvolil cestu, jejímž výsledkem je poměr jedné třídy implementující logiku dané agendy k jedné službě poskytované navenek serverem. Výsledkem je několik samostatných služeb jak je vidět na obrázku 4.2.

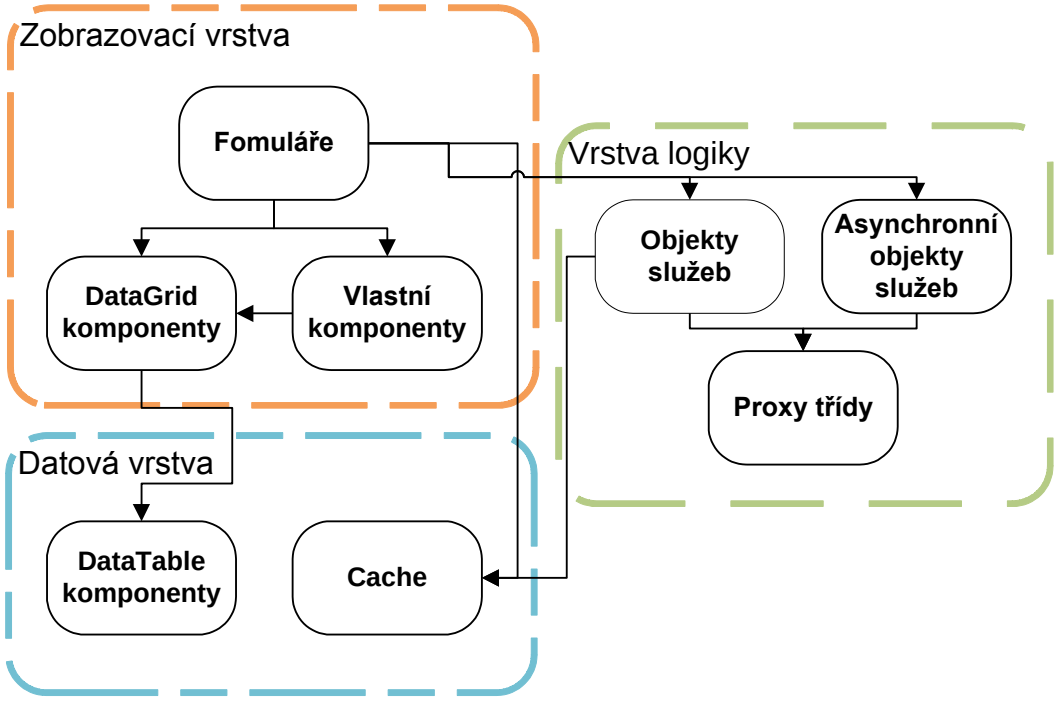


Obrázek 4.2: Celkový náhled na architekturu řešení

4.1.2 Klient

Většina formulářových aplikací je v dnešní době navrhována jako dvouvrstvá. Samozřejmě to není pravidlem, nicméně u většiny aplikací, se kterými jsem se setkal, to tak bylo. Horní vrstva je zobrazovací a ta pod ní je pak datová. Jedná se o jakousi mutaci návrhového vzoru MVC. Rozdíl je tvořen umístěním vrstvy Řadiče, která může spadat jak do datové vrstvy (Modelu) tak do zobrazovací vrstvy.(Pohledu). V mém případě jsem se nemusel nijak složitě rozhodovat, jak tento problém vyřeším, protože to za mě udělal WCF Framework. Díky automaticky generovaným proxy třídám (viz. 2.4.5) jsem „získal“ vrstvu Řadiče a celé řešení klientské aplikace se tak dá považovat za třívrstvé.

Z dosavadního textu nám zatím není jasné, co všechno se v jednotlivých vrstvách ukrývá a proto se podívejme na následující obrázek, který nám objasní tuto „záhadu“.



Obrázek 4.3: Pohled na „části“ klientské aplikace

Možná se ptáte, proč je modul datových tabulek (DataTable – viz. 4.5.1) ohraničen a označen v datové vrstvě, když jsme si v povídání o MVC architektuře řekli, že Model primárně slouží ke komunikaci s databází. Je to jednoduché. I přesto, že na klientské straně není využívána žádná databáze, nic nebrání tomu, abychom si definovali datovou vrstvu, která bude sloužit jako „uložiště“ příchozích dat.

Mimo datové tabulky se zde objevil i modul *Cache*, představující dočasné „skladiště“ dat, která se v čase nemění nebo mění, ale ne často. Proč byla využita tato neměnná paměť a k jakému účelu, se dozvíme v kapitole 4.3 a o něco později si v kapitole 4.5.1 ukážeme, jakým způsobem byly implementovány datové tabulky, které hrají roli „nosičů“ dat zobrazovaných tabulek na formulářích.

4.2 Omezení

Tato kapitola je věnována krátkému popisu několika omezení, jež s sebou přináší vývoj na mobilní platformě. Řeč bude o rychlosti práce s formuláři, objemu přenášených dat a jejich zpracování a na závěr kapitoly se zmíním o nepříjemných, leč stále častých výpadcích připojení do internetu.

4.2.1 Rychlost

Mobilní zařízení nedisponují tak velkým výkonem, který mají dnešní stolní počítače, proto je nutné s tímto faktem počítat i při návrhu implementace.

První kontakt s formuláři vedl k zamítnutí mé původní myšlenky, že systém práce s nimi bude obdobný jako u aplikací pro stolní počítače. Tedy vytvoření instance formuláře, zobrazení uživateli a po dokončení práce uživatele jeho zavření a odstranění z paměti. Jelikož je samotná inicializace formuláře paměťově a výkonnostně náročná, bylo nutné ji minimalizovat.

Vývoj probíhal na emulátoru, který je součástí vývojového prostředí Visual Studio 2008. Porovnáním jeho výkonu s dnešními mobilními zařízeními jsem došel k závěru, že jeho celkový výkon není nijak oslnivý. Emulátor by se svým výkonem dal zařadit mezi dnešní podprůměrná mobilní zařízení. Toto zjištění s sebou přineslo jednu malou výhodu – pokud bude klient implementován a optimalizován na pomalejší přístroj, tak na rychlejším bude pravděpodobně „běžet“ rychleji.

Minimalizace tvorby instancí formulářů a jejich předehrávání se stala předmětem kapitoly 4.3.1.

4.2.2 Objem přenosu dat

Požadavky na aplikace mohou být odlišné – někdo vyžaduje přehledné uživatelské rozhraní, rychlou reakční dobu nebo i paralelní zpracování informací. V našem případě je jedním z hlavních požadavků rychlost. Z předchozí kapitoly jsme se dozvěděli, že záleží na daném mobilním zařízení a jeho výpočetním výkonu. S tím souvisí i další problém, který se týká hlavně mobilních zařízení – objem přenesených dat.

Snaha minimalizovat objem přenášených dat je jasná – čím méně dat budeme odesílat / přijímat, tím rychlejší bude komunikace s „druhou stranou“. A u mobilních zařízení to platí obzvláště. Navíc je zde další element, který může vézt ke snaze snížit přenesený objem dat – jejich zpoplatnění.

Komunikace našeho řešení je založena na WCF Frameworku a jak jsme se dověděli z předchozích kapitol, WCF používá pro komunikaci SOAP zprávy založené na XML. Zasílané zprávy obsahují velké množství elementů a při přenosech větších množin objektů to může znamenat až stovky kilobajtů dat v jediné zprávě.

Klient při příjmu takto velkých zpráv musí vyhradit velké množství výpočetního výkonu pro jejich zpracování. To vede k časové prodlevě, která není žádoucí. V našem řešení proto byly přenášené objekty „zmenšeny“ (viz. kapitola 4.4) a musel být zaveden systém stránkování (viz. kapitola 4.5.3), redukující počet odesílaných entit.

Jednotlivé formuláře, hlavně ty, které jsou určené pro prohlížení a editaci jednotlivých entit, obsahují komponenty pro výběr hodnoty z nějaké množiny. Jedná se o roletková menu (ComboBox, DropDownList), které určitě znáte z webových a formulářových aplikací. Při zobrazování formulářů s detailem entit docházelo k přenosům stále stejných dat pro tato menu. Nejen, že to bylo plýtvání výpočetního výkonu, ale právě díky těmto přenosům došlo k výrazné prodlevě zobrazení formuláře. Důvodem bylo zpracování příchozí zprávy a vykreslení formuláře v jediném vlákně. Obdobně jako je řešen problém s formuláři, o němž jsme se dočetli

v předchozí kapitole, je implementováno i řešení problému zbytečného přenosu stále stejných dat. Tomu je věnována kapitola 4.3.2.

4.2.3 Výpadky připojení

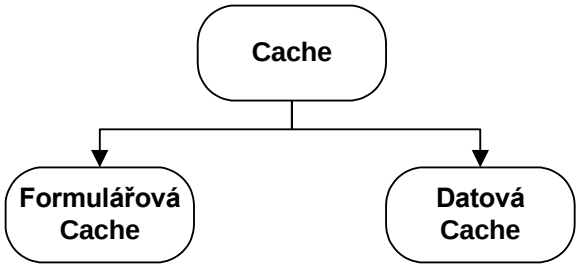
Několikrát padla zmínka o tom, že připojení do internetu na mobilních zařízeních nepatří mezi nejspolehlivější. Důvodem může být např. špatné pokrytí signálem, které vede k jeho ztrátě. Aby byl klient schopen pracovat i po výpadku připojení bylo nutné implementovat systém, který umožní, byť v omezené míře, klientu pracovat v odpojeném stavu. Řešení je věnována kapitola 4.7.

4.3 Cache

Taktéž známá pod pojmem *vyrovnávací paměť*, se využívá pro dočasné uložení dat. Jejím účelem je vyrovnání rozdílu rychlostí např. při předávání dat dvou prostředků. Příkladem může být pevný disk a operační paměť, kdy první jmenovaný hraje roli „pomalého“ a druhý „rychlého“ zařízení. Existuje *Cache* hardwarová a softwarová a v našem případě se budeme zabývat právě druhou jmenovanou.

Cache, tak jak jsme si ji krátce popsali, v našem případě ovšem nebude určena k vyrovnávání datových přenosů, nýbrž poslouží jako dočasné úložiště pro data a formuláře. Proto se nebude jednat o „pravou“ vyrovnávací paměť, ale o jakousi její „mutaci“.

Kapitola 4.2 poskytla nástin problémů, které je nutné vyřešit a nějakým způsobem optimalizovat. Jejich řešení založené na použití vyrovnávací paměti bude popsáno v této kapitole.



Obrázek 4.4: Dělení vyrovnávací paměti

4.3.1 Správa formulářů

Jak napovídá obrázek 4.4, vyrovnávací paměť je rozdělena na dva oddíly. První z nich je určena k ukládání instancí formulářů a jejich správě. Druhému oddílu je věnována následující kapitola.

Proč že to vlastně budeme ukládat formuláře a využívat jen jednu instanci každého z nich? Existuje hned několik odpovědí. Hlavním argumentem jsou relativně velké nároky na „výrobu“ formuláře.

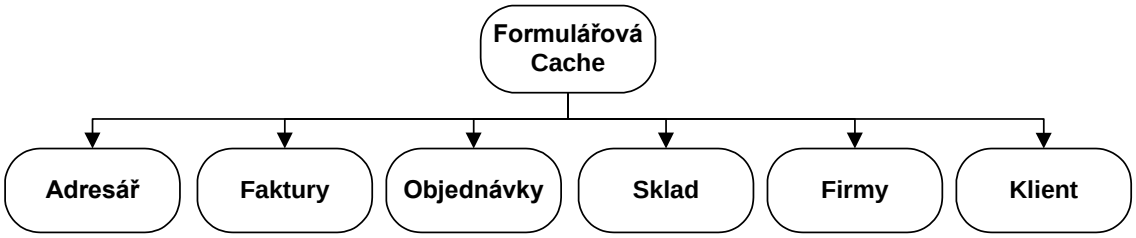
O čem jsem se doposud nezmínil, je fakt, že otevřený formulář se ve výchozím nastavení nezavírá. Jinými slovy – instance formuláře, nad kterým pracujeme, po označení objektu pro zavření, křížku, není zlikvidována. To znamená, že není uvolněna z paměti a namísto tohoto očekávaného chování, je formulář „schován“. Operační systém si „drží“ jeho stav a udržuje jej na živu během práce s aplikací.

Toto chování nám „nahrává do karet“ a díky němu stačí zavést mechanismus, který bude „držet“ odkaz na náš již existující a schovaný formulář. Díky drženému odkazu jsme schopni znovu formulář použít, zobrazit jej a hodnoty v něm případně měnit podle potřeby.

Třídy, které se starají o úschovu odkazů na formuláře jsem nazval *FormHolder*. Ještě než se podíváme na ukázku kódu jejich implementace, zmíním ještě jednu vlastnost operačního systému Windows Mobile.

Pokud jste se někdy dostali do „křížku“ s tímto operačním systémem, určitě vám neušla skutečnost, že není možné mít aktivních více formulářů najednou. To znamená, že v jednu chvíli můžete mít zobrazen jen jeden formulář. Ty zbylé, které jste si mohli otevřít předtím, jsou tím aktuálním „překryty“. Takže pokud si otevřete libovolný formulář a hned poté např. mobilní webový prohlížeč, dojde k překrytí prvního formuláře. Jakmile však ukončíte práci s webovým prohlížečem a zavřete jej, „objeví“ se předtím překrytý formulář.

Zmíněné vlastnosti tedy neumožňují práci na dvou stejných formulářích, což usnadnilo návrh tříd uchovávající si odkazy na otevřené formuláře.



Obrázek 4.5: Dělení vyrovnávací paměti na agendy

Jak ukazuje obrázek 4.5, dočasný úložný prostor je rozdělen podle agend. Jednotlivé *FormHoldery* pro každou s agend uchovávají všechny formuláře pod ní spadající. Např. *FirmaFormHolder* má na starost veškeré formuláře týkající se firem – hlavní formulář, detail firmy, výběr odběratele a dodavatele, formulář detailu osob a poboček firmy.

Každá z těchto tříd je implementována podle návrhového vzoru *Jedináček*. (Singleton) Návrhový vzor *Jedináček* spadá do rodiny vzorů *Tvořících* (Creational) a jeho cílem je zajistit existenci jediné instance třídy. Ve výsledku to znamená, že máte k dispozici jediný přístupový bod k funkcionalitě, kterou třída poskytuje. Implementace *Jedináčka* se taktéž využívá i v jiných návrhových vzorech např. Fasáda, Abstraktní továrna a jiné. Podrobnější informace jsou k nalezení zde [8].

```
public sealed class FirmaFormHolder : IFormHolder
{
      private static readonly FirmaFormHolder _instance = new
      FirmaFormHolder();

      private FirmaForm _firma;
      public FirmaForm FirmaForm {  get { return _firma; } }

      private FirmaFormHolder()
      { }

      public static FirmaFormHolder Instance
      {
            get { return _instance; }
      }
      //další kód třídy
}
```

Obrázek 4.6: Implementace *Jedináčka* pro třídu obsluhující formuláře firem

Práce s třídou *FormHolderu* je velmi jednoduchá. V našem případě stačí zavolat veřejnou statickou vlastnost *Instance*, představující objekt daného *FormHolderu*. Na něm lze volat další metody a vlastnosti. Z ukázky je vidět, že díky vlastnosti *FirmaForm* máme přístup k uchovávanému formuláři firmy, který je používán pro zobrazení jejího detailu. Právě nastavení hodnot detailu firmy do formuláře a jejího zobrazení ukazuje následující řádek kódu.

```
FirmaFormHolder.Instance.FirmaForm.SetObjectAndShow(firma);
```

4.3.2 Uchovávání neměnících se informací

Správa formulářů představovala jednoduchou paměť, jejímž cílem bylo poskytovat formuláře k znovupoužití. Dočasná paměť, o které bude řeč nyní, je trošku komplikovanější. Taktéž využívá návrhového vzoru *Jedináček*, ale mimo to, ještě přidává novou funkcionalitu podobnou návrhovému vzoru *Pozorovatel*.[9]

Potřeba dočasného uchování opakujících se informací vedla k vytvoření třídy, díky níž bude minimalizován datový přenos mezi serverem a klientem. Primárně tato třída slouží k úschově dat z tzv. „číselníků“, což jsou tabulky v databázi na straně serveru, obsahující data, k jejichž změnám dochází velmi málo nebo vůbec ne. Na straně klienta jsou tato data zobrazována v roletkových menu, o kterých byla zmínka v kapitole 4.2.2.

Jelikož je komponenta roletkového menu velmi jednoduchá a nenabízí speciální zobrazování či jinou pokročilou funkcionalitu, bylo nutné provést několik malých modifikací, které vedly k implementaci vlastní komponenty s názvem *AdvancedComboBox*.

Visuálně nedošlo k žádným změnám, pouze bylo dodáno několik metod a vlastností do již existující komponenty roletkového menu. Stěžejní vlastností je *Type* výčtového datového typu *CBObjectType*, který může nabývat hodnot:

- Zakazka
- Zeme
- Cenova_skupina
- Pobocka
- Firma
- Mena
- Sklad
- KategorieSkladu

Jednotlivé hodnoty identifikují použití roletkového menu. Jinými slovy, pokud bychom měli menu s vlastností *Type* nabývající hodnoty *Zeme*, data která bude obsahovat, budou představovat seznam zemí. Výčtový typ *CBObjectType* tak slouží i jako identifikátor dat, přicházejících ze serveru na klienta.

Vraťme se ale zpět k hlavní třídě naší dočasné paměti. Ta obsahuje dynamické datové struktury k uchovávání příchozích dat, která jsou identifikována právě výčtovým typem *CBObjectType*. Mimo to, má třída také k dispozici struktury pro ukládání objektů typu *AdvancedComboBox*. V zápětí objasním proč.

Třída dočasné paměti na sebe bere zodpovědnost za obsah dat v roletkových menu na formulářích. Aby toho byla schopna, bylo nutné implementovat mechanismus „registrace“. Jednotlivé menu se registrují u třídy dočasné paměti a ta si uchová jejich odkaz. Díky tomu může dojít k naplnění jednotlivých menu daty v závislosti na výčtovém typu *CBObjectType*.

Samotná registrace jednotlivých roletkových menu vypadá následovně:

```
CBDataCache.Instance.AddCacheDataListener(acbZeme);
```

Přístup je obdobný jako u třídy správy formulářů. Možná vás napadla otázka, jakým způsobem vlastně třída *CBDataCache* získává data. Odpověď nám dá definice následující metody:

```
public void SetCacheData(GenericCBObject[] data)
```

Tuto metodu je možné volat prakticky kdykoliv po inicializaci formulářů a komponent, pokud tedy máme k dispozici nějaká data. Jakmile však data předáme, dojde k následující posloupnosti událostí:

1. Všechna roletková menu jsou deaktivována
2. Je v nich zobrazena informace o nahrávání nových dat
3. Jsou nahrána nová data
4. Všechna roletková menu jsou aktivována

Další otázka, která vás teď mohla napadnout je „co když se data na serveru změní?“. Možností jak na tuto otázku odpovědět je více a věřím, že vás napadají různá řešení. To mé se pokusím krátce popsat i přesto, že zatím nebylo do „ostré“ verze nasazeno.

První nutnou věcí je stanovit si interval představující planost dat. Tato volba je zatím řízena konstantou, nicméně v budoucnu bude jistě přenechána k nastavení uživateli. Tato

konstanta tedy udává délku platnosti dat v roletkových menu. Jakmile vyprší, musí dojít k znovu nahrání dat ze serveru. Tento mechanismus má na starost metoda *CheckCacheData*, jejíž implementace je zobrazena níže. Zásadní roli zde hraje i použití tzv. časového razítka (timestamp), což není nic jiného než datum a čas. Při každém přiřazení dat, je vždy nastaveno aktuální časové razítko.

```
public void CheckCacheData()
{
 if(this._timestamp.
      AddMinutes(ClientConfiguration.CACHE_CB_DATA_EXPIRATION_TIME)<
      DateTime.Now)
  {
    SetLoadingStateToListeners();
    AsyncCBObjectServiceObject.Instance.GetListAll(this);
  }
  //v opacnem pripade stavajici data zustavaji a jsou platna
}
```

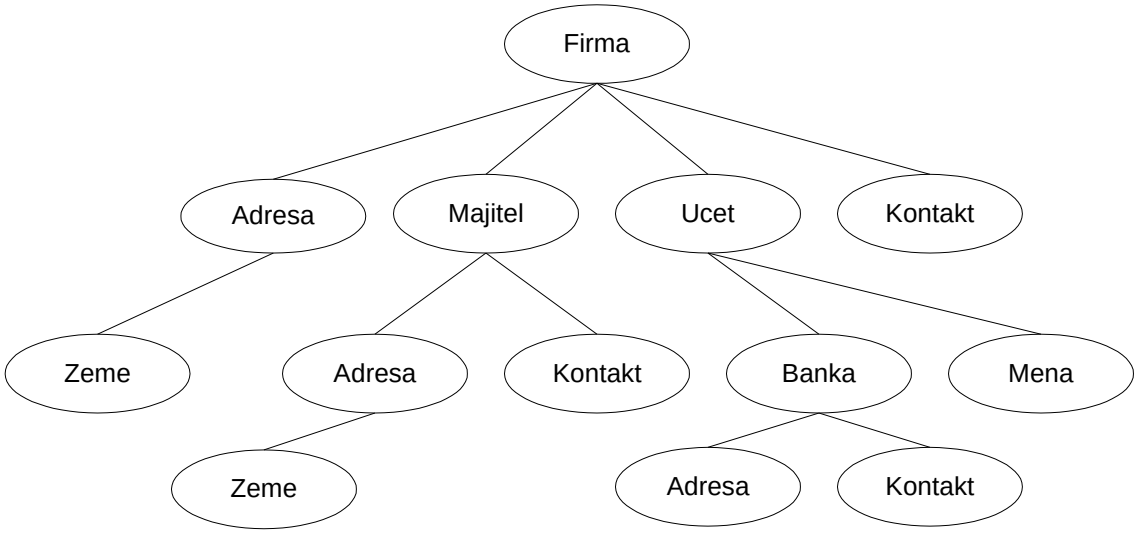
Obrázek 4.7: Implementace metody CheckCacheData

Jediné co zde zůstává nezodpovězeno, je použití metody *CheckCacheData*. Naskýtá se totiž otázka, kde by měla být volána. Jelikož se celý problém okolo aktualizace dat v dočasné paměti točí kolem roletkových menu, volba v mém případě padla do části kódu, který zobrazuje jednotlivé formuláře.

4.4 Zjednodušení objektů

Objekty reálného světa jsou zřídka kdy jednoduché. Jejich „převedení“ do libovolného objektově orientovaného programovacího jazyka může být ještě obtížnější. Čím bude objekt složitější, tím košatější bude jeho objektový strom. Záleží však také na objektovém návrhu.

Nyní se dostávám k problému přenosu velkých objektů za pomocí WCF Frameworku na mobilní zařízení. Po lehkém úvodu do této technologie víme, že zprávy zasílané ze serveru na klienta obsahují velké množství XML elementů. Pro připomenutí doporučuji prohlédnout si obrázky z kapitoly 2.4.3, konkrétně 2.7 a 2.8. Na těch je zobrazen kód požadavku a odpovědi na dotaz pro zjištění počtu osob. Návratová hodnota odpovědi zabírá zlomek toho co veškeré XML kolem. A teď si představte, jak bude vypadat odpověď vracející množinu firem, jejichž zkrácený objektový strom je zobrazen na obrázku 4.6.



Obrázek 4.6: Zredukovaný objektový strom firmy

Určitě je vám jasné, že přenos tak velkého množství dat je nutné nějakým způsobem zredukovat. Ať už z důvodu ušetření datového přenosu nebo zbytečnosti některých informací. Celý problém se týká hlavně zobrazení informací v tabulkách na straně klienta, kdy je potřeba přenést více než jednu entitu.

Musíme si položit otázku, které atributy bude člověk využívající mobilního klienta, považovat za důležité a které ne. Navíc, čím méně jich bude, tím lépe. Výsledkem bude redukovaná množina čítající několik atributů, jako v případě mé redukce informací u firmy.

```
[DataContract]
public class FirmaGridObject
{
    [DataMember]
    public int Id { get; set; }
    [DataMember]
    public string Cislo_firmy { get; set; }
    [DataMember]
    public string IC { get; set; }
    [DataMember]
    public string DIC { get; set; }
    [DataMember]
    public string Nazev { get; set; }
}
```

Obrázek 4.7: Redukovaná třída firmy

Z původních cirka osmdesáti atributů, jsem vybral pět nejpodstatnějších. Ale stále ještě zbývá vyřešit problém s kompletním zasíláním množin entit. V navazující kapitole si proto představíme různé komponenty použité na formulářích. Mezi ně patří i komponenta *Pager*, určená ke stránkování záznamů.

Zjednodušení našlo uplatnění i pro objekty používané v roletkových menu z kapitoly 4.3.2. Díky jednoduchosti grafického rozhraní těchto komponent není potřeba více než jednoho atributu. Jím byl ve většině případů název entity, který je roletkovým menu zobrazován uživateli. Pro rozlišení unikátnosti hodnot ale jeden atribut nestačí, a proto bylo do objektu spolu s názvem zahrnuto i identifikační číslo a typ dat. (viz. kapitola 4.3.2)

```
[DataContract]
public class GenericCBObject
{
    [DataMember]
    public int Id { get; set; }
    [DataMember]
    public string Nazev { get; set; }
    [DataMember]
    public CBObjectType Type { get; set; }
}
```

Obrázek 4.8: Obecná třída pro data určená roletkovým menu

4.5 Komponenty

.NET Compact Framework nabízí velké množství komponent použitelných pro různé účely, i přesto se však stávalo, že nabízená množina nebyla schopna pokrýt veškeré mé požadavky. Proto bylo nutné implementovat si komponenty vlastní. Ve většině případů se jednalo o deriváty existujících komponent. Ty byly obohaceny o funkcionalitu, která se hodila ať už pro řešení problému či usnadnění práce.

Vlastní komponenty implementované a použité v řešení by se daly rozdělit do tří skupin. První z nich jsou „neviditelné“ komponenty, jejímž jediným zástupcem je *DataTable*[13]. Proč neviditelné? Jednoduše proto, že nedisponují žádným grafickým rozhraním. Na druhé straně jsou komponenty s grafickým rozhraním, jako např. *DataGrid*. Do jejich podmnožiny spadají komponenty složené (kompozitní) mající v našem řešení dva zástupce, a to *Pager* a několik speciálních tabulkových zobrazovacích komponent, jimž jsem dal název *GridControls*.

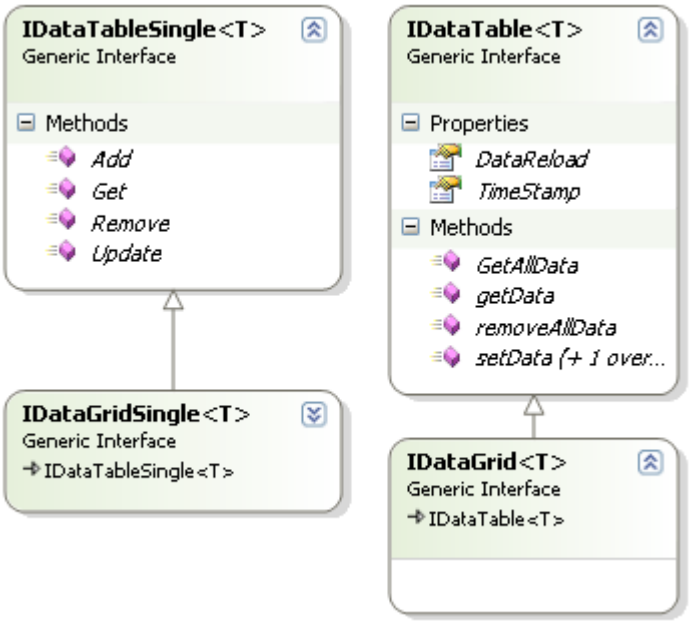
4.5.1 DataTable

Třída *DataTable* představuje jednu tabulku pro uložení dat v paměti. Můžeme si ji představit jako obdobu tabulky, kterou známe z relačních databází. Datová tabulka může mít definován primární klíč, názvy atributů a obsahovat záznamy.

DataTable tedy slouží jako jasně definované uložiště dat, které mohou využívat další komponenty. Ve většině případu se jedná o komponenty tabulek s grafickým rozhraním. V našem řešení tomu není jinak a taktéž jsou datové tabulky použity jako vrstva pod komponentami zobrazujícími jejich data.

I přesto, že DataTable disponuje velkým množstvím funkcionality, jsem se rozhodl tuto třídu modifikovat a ulehčit si práci přidáním několika metod, které usnadní přístup k datům.

Toho jsem docílil definicí generického rozhraní, které všechny datové tabulky (DataTable) implementují.

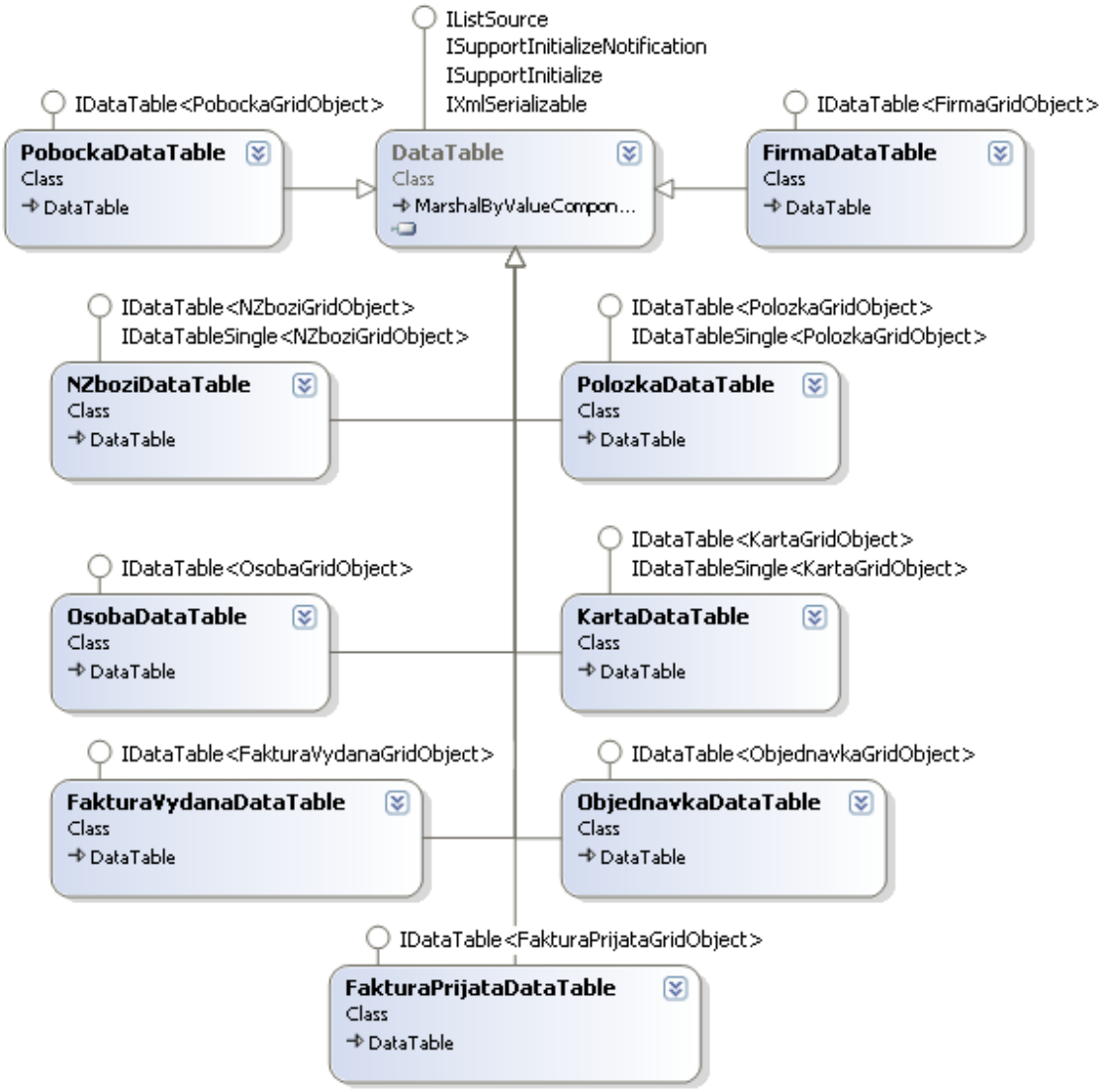


Obrázek 4.9: Dvojce rozhraní pro datové tabulky

Obrázek 4.9 zobrazuje dvě rozhraní, z nichž dědí jiná rozhraní určená pro komponenty tabulek, se kterými nás seznámí kapitola 4.5.2. Jak bylo zmíněno, všechny datové tabulky implementují rozhraní *IDataTable<T>* poskytující metody pro jednoduchou práci s daty. Na obrázku je však ještě druhé rozhraní *IDataTableSingle<T>* definující čtyři základní operace – přidání, získání, odstranění a modifikaci objektu.

Toto rozhraní bylo později přidáno z toho důvodu, že původní rozhraní *IDataTable<T>* nepočítá s operacemi nad jednotlivými záznamy dané tabulky. Operace z rozhraní *IDataTableSingle<T>* nejsou potřeba u všech tabulek a proto bylo toto rozhraní odděleno.

Z povídání výše vyplynulo, že řešení obsahuje více datových tabulek. Které to jsou, ukazuje následující obrázek.



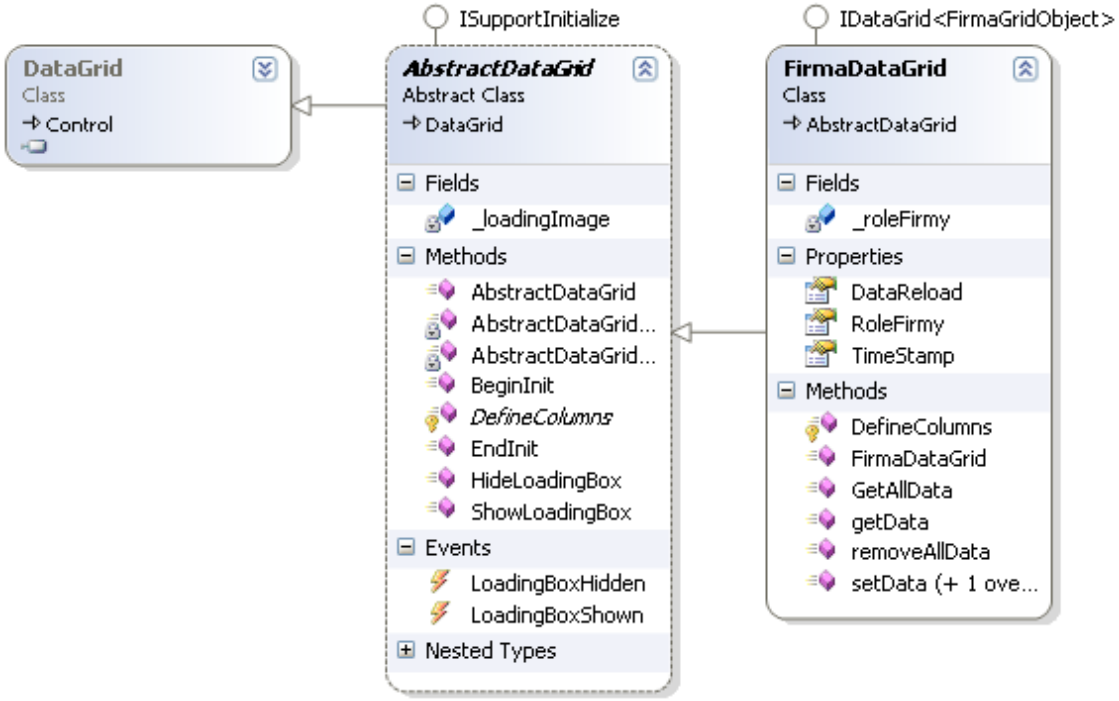
Obrázek 4.10: Třídní diagram datových tabulek

Jednotlivé datové tabulky, pro každou z agend, dědí svou funkcionalitu od „originální“ datové tabulky z .NET Compact Frameworku. V množině datových tabulek na obrázku se vyskytují i takové, které nejsou přímo „zástupcem“ agendy, ale jsou jakousi „pomocnou“ silou při řešení její funkcionality. Příkladem může být *PolozkaDataTable*, která je využita jako podkladová vrstva pro tabulky zobrazující položky jednotlivých dokladů – faktur a objednávek. Implementace rozhraní *IDataTableSingle<PolozkaGridObject>* poté dává možnost práce s jednotlivými položkami dokladů, kdy můžeme měnit jejich atributy a přidávat je např. ze skladu či čtečky (viz.4.6.5)

4.5.2 DataGrid

Jak název napovídá, bude se jednat o komponentu pro zobrazovaní dat v nějaké mřížce – tabulce. Stejně jako .NET Compact Framework nabízel k použití *DataTable*, je k dispozici i tato komponenta. Rozdíl oproti *DataTable* je v tom, že *DataGrid* disponuje grafickým rozhraním a uživatel tak vidí zobrazená data.

Originální *DataGrid* implementovaný firmou Microsoft nabízí mnoho možností konfigurace a funkcionality. Mé řešení přebírá veškeré tyto vlastnosti a schopnosti a přidává několik specialit. Mezi ně bych zařadil zobrazení stavu, že dochází k nahrávání dat pro tabulku. Tato funkcionalita byla implementována v rodiči ostatních vlastních *DataGridů.*



Obrázek 4.11: Třídní diagram pro tabulku zobrazující firmy

Tím rodičem je abstraktní třída *AbstractDataGrid*, dědící veškerou funkcionalitu originálního *DataGridu*. Abstraktní třída byla obohacena o funkci zobrazení obrázku, indikující nahrávání dat. Poskytuje také dvě události *LoadingBoxHidden* a *LoadingBoxShown*, které mohou být využity k různým účelům. Události jsou spouštěny, jakmile dojde ke schování respektive zobrazení nahrávacího obrázku. Třída dává možnost zobrazování těchto obrázku ručně řídit pomocí metod *HideLoadingBox* a *ShowLoadingBox*. Abstraktní metoda *DefineColumns*, jak název napovídá, slouží k definici jednotlivých atributů (sloupců) pro tabulku, kdy každý potomek této třídy definuje vlastní sloupce, které jsou brány v potaz při inicializaci komponenty.

Zmínil jsem fakt, že je nutné definovat atributy (sloupce). Je to z toho důvodu, aby *DataGrid* věděl jaké bude zobrazovat sloupce a hlavně jak budou mapovány na data z *DataTable* komponenty. Ta definuje množinu atributů a uchovává data. Aby celý mechanismus fungoval, je nutné definovat vztah mezi příslušnou *DataTable* a *DataGridem*.

Definice vazby mezi těmito komponentami je jednoduchá a ukážeme si ji na příkladu mapování *PobockaDataTable* na *PobockaDataGrid*.

```
public PobockaDataGrid(PobockaDataTable dt):this()
{
     this.TableStyles[0].MappingName = dt.TableName;
     this.DataSource = dt;
     this.Refresh();
}
```

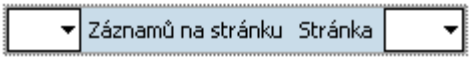
V ukázce je zobrazen konstruktor *PobockaDataGridu*, tedy tabulky zobrazující záznamy poboček k dané firmě. Samotné přiřazení datové tabulky se děje na prvním řádku. Na řádku následujícím je zobrazovací tabulce přiřazen datový zdroj - parametrem zasílaná *PobockaDataTable* s daty.

Velmi podobně jak je implementován *PobockaDataGrid*, jsou implementovány i ostatní *DataGridy*. Ještě než přejdeme k další kapitole, zbývá poznamenat, že většina *DataGridů* se stala základem pro kompozitní (složenou) komponentu, kterou jsem pojmenoval *GridControl* o níž bude řeč v kapitole 4.5.4.

4.5.3 Pager

Během seznamování s problémy v kapitole 4.2 jsem několikrát odkázal na komponentu, která částečně pomůže při řešení problému přenosu většího množství objektů. Tato komponenta nese název *Pager*. Jedná se o kompozitní ovládací prvek, složený ze standardních komponent, které nabízí .NET Compact Framework.

O jaké ovládací prvky se jedná určitě poznáte z následujícího obrázku.



Obrázek 4.12: Komponenta Pager

Co vlastně samotný Pager dělá? Jeho úkolem je informovat o zvolené stránce a počtu záznamů. Volbu umožňují dvě roletková menu po stranách komponenty. Výchozí hodnoty počtu záznamu na jednu stránku jsou 10 20 a 50. Díky tomuto omezení nedochází k přenosu celých množin objektů a mobilní zařízení není tak zatíženo zpracováním velkého množství dat. Roletkové menu pro selekci stránek je naplněno při iniciliazaci formuláře a jeho hodnoty se mohou v průběhu práce měnit.

Informace o změnách hodnot v roletkových menu komponenty jsou předávány za pomocí dvou událostí:

- PageChanged
- PageSizeChanged

Událost *PageChanged* – změna stránky, je odesílána ve chvíli kdy dojde k přepnutí na jinou stránku. Reagovat na událost *PageChanged* lze různě, nicméně nejvhodnějším řešením je implementace obslužné metody, která se postará o nahrání nových dat na základě zvolené stránky. Druhá z událostí - *PageSizeChanged*, je odesílána pokud dojde ke změně velikosti stránky. I zde je vhodná implementace metody, která zajistí získání nových dat na základě zvolené velikosti.

Je jasné, že Pager nemůže existovat samostatně, respektive může, ale nemělo by to žádný význam. Proto došlo k „sloučení" komponent *Pager*, *DataGrid* a vznikla nová komponenta s názvem *GridControl*. Právě díky *Pageru* jsme schopni reagovat na změny uživatele a tyto změny promítnout do zobrazení dat v komponentě *DataGrid*.

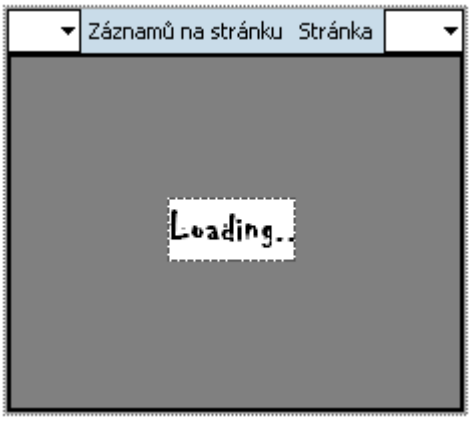
4.5.4 GridControls

Jak už bylo několikrát řečeno, *GridControl* je komponenta tzv. kompozitní. Znamená to, že je složena z několika již existujících komponent, ať už původních z .NET Compact Frameworku či vlastních.

Následující seznam komponent tvoří jádro *GridControlu*:

- Pager
- DataGrid (vlastní)

Po představení těchto „stavebních bloků", máme představu jaká je jejich funkcionalita a proto se ještě podívejme na výsledek jejich spojení.



Obrázek 4.12: Komponenta GridControl

Pro každou z podporovaných agend byl implementován jeden *GridControl*. Samotná komponenta představuje velkou část logiky na hlavních formulářích, na kterých je vesměs zobrazena jen ona. Aby mohl uživatel nějakým způsobem pracovat s jednotlivými záznamy tabulky, každá z nich má vlastní kontextové menu. Obsah kontextového menu je závislý na dané tabulce. Např. u *PolozkaGridControl* komponenty neexistuje možnost vložit nový záznam

z kontextového menu. Je to dáno tím, že přidávání položek na doklady je řešeno pomocí dvou speciálních tlačítek. Více v kapitole 4.6.2 a 4.6.3.

Kontextová menu obsahují tyto prvky:

- Nová
- Detail
- Obnovit
- Odstranit

Výskyt těchto prvků, jak jsem říkal, záleží na daném *GridControlu*, popřípadě *DataGridu*. Z názvů je patrné jaký mají význam. Snad jen *Obnovení* by mohlo vyvolat určité pochybnosti. Jedná se o aktualizaci dat v dané tabulce (*DataGridu, GridControlu*), což znamená, že je odeslán požadavek na server pro nahrání dat, jejichž rozsah a počet je ovlivněn nastavenými roletkovými menu v komponentě *Pager*.

Kromě spojení funkcionalit komponent zmíněných výše byly do jednotlivých *GridControls* přidány vlastnosti potřebné např. k identifikaci obsahu. Příkladem může být *ObjednavkaGridControl*, jež může obsahovat objednávky přijaté a objednávky vydané, nebo *FirmaGridControl* a jeho vlastnost *RoleFirmy*, která rozděluje firmu na odběratele, dodavatele a výrobce.

Abych to shrnul, *GridControl* je vylepšená tabulka určená k zobrazování dat, disponující vlastní logikou. Ta se stará o získávání dat ze serveru v závislosti na uživatelském vstupu, který poskytuje její část – *Pager*. Během nahrávání dat je uživateli zobrazena informace o průběhu a ten tak získává zpětnou vazbu o tom, že se „něco“ děje.

4.6 Formuláře

Mobilní klient podporuje celkem pět agend – fakturace, objednávky, sklad, identifikace a firmy. Každá ze jmenovaných má v řešení jednoho „hlavního“ formulářového zástupce. Jinými slovy, hlavní obrazovku, ze které se dají provádět další operace.

Kapitola popisuje řešení formulářů pro jednotlivé agendy včetně jejich funkcionality. Doprovodné obrázky jsou určeny pro jasnější představu o vzhledu celé aplikace.

4.6.1 Adresář

Agenda adresáře slouží pro snadný přehled a přístup k informacím o firmách uložených v ERP systému. Většinou to bývají firmy dodavatelů, odběratelů, výrobců ale i jiných. Mobilní klient podporuje zobrazení právě těchto tří typů firem.

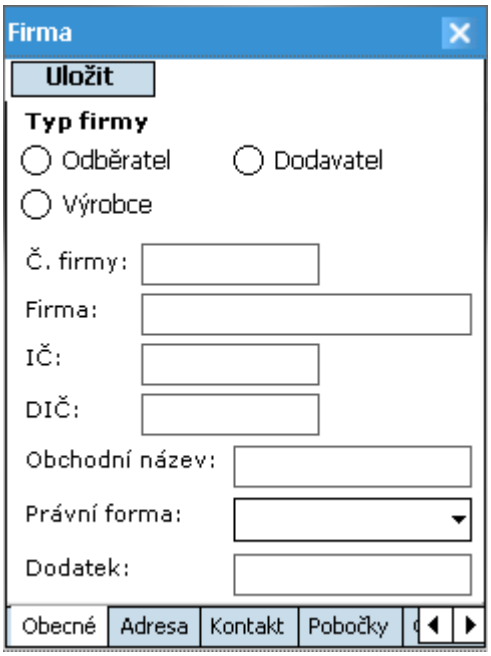
Mobilní zařízení disponuje relativně malou plochou, na které se formuláře zobrazují. Aby bylo možné zobrazit tolik informací, jakými jsou seznamy různých typů firem, bylo nutné využít komponentu záložek. Ta poskytuje několikanásobně větší prostor pro zobrazení prvků na formuláři za cenu jejich přepínání.

Hlavní formulář obsahuje tří záložky, kdy každá z nich je věnována jednomu typu firmy. Na jednotlivých záložkách se nachází další komponenta, o které byla řeč v kapitole 4.5.4

– *GridControl*. V našem případě *FirmaGridControl*. Díky přidaným vlastnostem, mezi které patří *RoleFirmy*, ji nebylo nutné implementovat pro každý typ firmy zvlášť.

Kontextová menu, která byla zmíněna dříve, dávají možnost uživateli provádět různé operace. První z nich je přidání nového záznamu firmy. Jakmile si uživatel tuto možnost zvolí, je mu zobrazen formulář pro zadání údajů o firmě. Ten taktéž využívá záložek pro rozšíření zobrazovacího prostoru. Po vyplnění údajů může uživatel firmu uložit, kdy jsou informace zasílány na server v případě dostupného připojení. V opačném případě je entita uložena do specifického adresáře nacházejícího se v adresáři aplikace. (viz. kapitola 4.7)

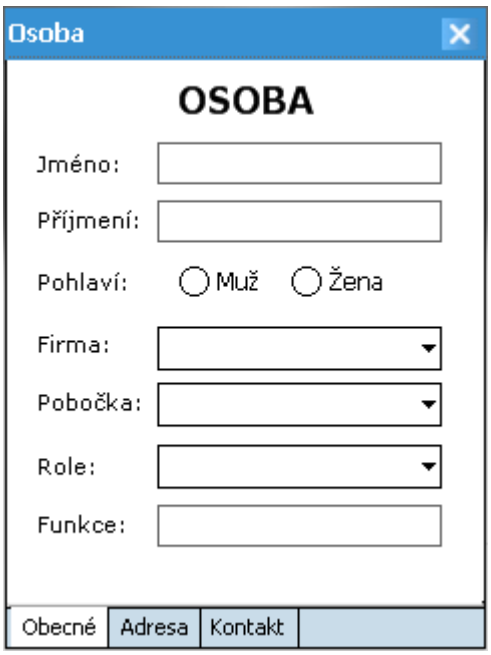
Ve chvíli kdy se chce uživatel podívat na detail některé z firem, postačí z kontextového menu zvolit možnost *Detail*. V ten moment je zobrazen identický formulář jako v případě vkládání nového záznamu. Během prohlížení detailu existuje možnost záznam měnit a platí všechno, co platí u vkládání nového záznamu.



Obrázek 4.13: Formulář firmy

Jak vypadá formulář zobrazovaný při vkládání, či detailu záznamu vidíte na obrázku 4.13. Mimo standardní záložku *Obecné*, *Adresa* a *Kontakt* byla přidána i záložka se seznamem poboček dané firmy. Možnosti operovat s pobočkami firmy jsou však omezenější a není k dispozici např. přidávání nové pobočky k firmě. Nicméně možnost prohlížet si její detail zůstala. Formulář detailu prohlížené pobočky je podobný tomu na obrázku 4.13 a mimo standardní záložky *Obecné*, *Adresa*, *Kontakt* obsahuje i záložku *Osoby*. Na té se nachází tabulka, v níž jsou záznamy o osobách pracujících na dané pobočce.

Tak jako tomu bylo u firmy a poboček, zvolením detailu osoby je zobrazen formulář poskytující základní informace o dané osobě. Ani zde však není možnost, přiřazovat nové osoby k pobočce.



Obrázek 4.14: Formulář osoby

4.6.2 Fakturace

Vzhled hlavního formuláře fakturací se ubral stejným směrem jako formulář adresáře. Je velmi jednoduchý a na dvou záložkách, které dělí faktury na vydané a přijaté, jsou uživateli zobrazeny tabulky s jejich záznamy.

Taktéž operace nad jednotlivými tabulkami jsou obdobné jako u adresářových tabulek. Formulář s detailem faktury obsahuje záložky *Obecné*, *Odběratel*/*Dodavatel* (záleží na typu faktury), *Položky*, *Souhrn* a *Ostatní*. První dvě jmenované obsahují informace o dokladu samotném a firmě, na níž je doklad vystaven.

Doklad faktury či objednávky by neměl význam, pokud by neobsahoval nějaké položky. Přehled nám o nich dává záložka *Položky,* na které je zobrazena tabulka s konkrétními položkami dokladu. Získáváme tak informaci o jejich cenách, názvech, množství. Jednotlivé položky lze editovat a díky dvěma tlačítkům umístěným pod tabulkou je lze na doklad přidávat. Možnosti jsou dvě – vybrat si nějaké zboží ze skladu nebo „načíst“ zboží z čtečky čárových kódů. Jak vypadají formuláře obou možností lze vidět v kapitole 4.6.4 a 4.6.5.

Obrázek 4.15: Formulář faktury přijaté

Záložka *Souhrn* poskytuje informace o celkové sazbě DPH, počtu položek, počtu kusů jednotlivých produktů, celkové ceny bez a s DPH. Díky tomuto souhrnu má uživatel jasný přehled o vystavované respektive přijaté ceně dokladu.

4.6.3 Objednávky

Další podporovanou agendou jsou objednávky. Uživatel je může přímo vytvářet z terénu a zasílat do „velkého“ systému pro další zpracování.

Uživatelské rozhraní, respektive formuláře jsou navrženy prakticky stejně jako formuláře faktur. Až na drobné rozdíly v atributech dokladů se jedná o tytéž formuláře. (viz. obrázek 4.15) Převzata byla i funkcionalita, kterou jsme si krátce představili v předchozí kapitole. Taktéž padla zmínka o dvou speciálních tlačítcích na záložce *Položky*. Co se stane a co se uživateli zobrazí po jejich volbě si ukážeme v následujících dvou kapitolách.

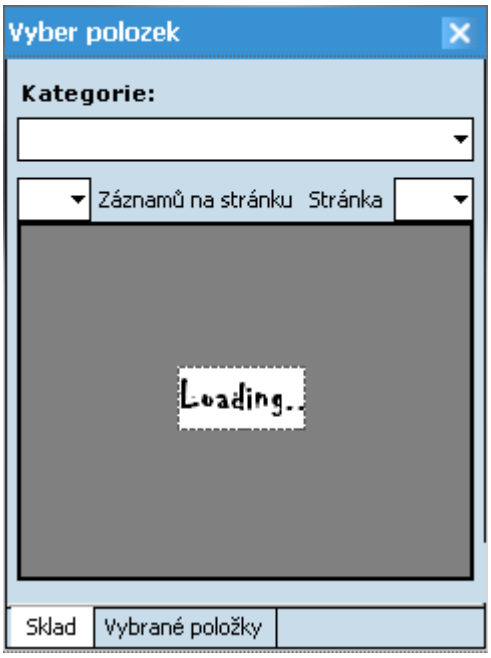
4.6.4 Sklad

Firma, která vyrábí nějaké výrobky či jen prodává zboží ve většině případů disponuje i nějakým skladem. Rozumným krokem je vézt si záznamy o stavu skladových zásob, výdejích a příjmech zboží. To jsou základní operace, které by měla firma v rámci své existence nějakým způsobem řešit. Existují systémy, které řízení skladu ulehčují a zpřehledňují. Ve větších systémech se jedná o moduly, mezi které spadá i skladové hospodářství. Jelikož systém nad kterým pracuje

mobilní klient takovýto modul má, je důležité, aby jej alespoň z části podporoval i mobilní klient.

Řešení nepočítalo s vícero sklady i když to většinou bývá tak, že firmy mají více než jeden sklad. V našem případě uvažujeme pouze jediný sklad, který pokrývá všechny zásoby firmy. Jednotlivé položky skladu jsou děleny do kategorií. Ty mohou být různé, podle zaměření dané firmy. Jejich vytváření však není možné realizovat z mobilního zařízení a tato možnost je dostupná jen z ERP systému.

Formulář skladu je tvořen z několika komponent, z nichž některé byly představeny v kapitolách 4.5.2 a 4.5.3 respektive 4.5.4. Uživatelské rozhraní je zobrazeno na obrázku 4.16.



Obrázek 4.16: Formulář výběru položek ze skladu na doklad

Grafické rozhraní je velmi jednouché. Pomocí roletkového menu si uživatel vybere kategorii, která jej zajímá, což zapříčiní nahrání prvních několika záznamů ze serveru do tabulky. Díky stránkovací komponentě můžeme tyto záznamy v rychlosti prohlížet. Tabulka, ačkoliv to na obrázku není vidět, poskytuje tyto informace – *Číslo karty*, *Název*, *Skladem*. Poslední jmenovaný atribut je vyjádření počtu zboží na skladě.

Záměrně jsem namísto původního obrázku formuláře skladu předložil formulář pro výběr položek. Rozdíl mezi ním a formulářem skladu je totiž minimální. Jediným rozdílem je existence záložek, respektive záložky *Vybrané položky*. Když si ji odmyslíte, dostanete vzhled formuláře pro sklad. Ale proč jsem to vlastně ukázal formulář pro výběr položek?

Pokud si vzpomínáte, v kapitole 4.6.2 se na obrázku 4.15, kde byl zobrazen formulář dokladu faktury objevily dvě tlačítka. Jednalo se o tlačítka pro přidávání položek na doklad. Pokud uživatel zvolil první z nich – *Přidat ze skladu* – byl mu zobrazen právě formulář z obrázku 4.16. Dříve padla zmínka o kontextových menu, i výběr položek na doklad jedno takové má. Mezi jeho položky patří operace *Přidat*. Pokud tedy chceme přidat konkrétní zboží

ze skladu na doklad, stačí jej označit v tabulce, vyvolat kontextové menu a zvolit operaci *Přidat*. Jakmile k tomu dojde, zboží je přidáno do tabulky na druhé záložce z obrázku 4.16 – *Vybrané položky*. Pokud bychom si rozmysleli přidání některé z položek, je možné záznam odstranit, opět skrze kontextové menu. Tlačítkem pod tabulkou vybraných položek uživatel potvrdí jejich volbu a ty jsou „přeneseny“ na doklad ze kterého byl výběr položek vyvolán.

Určitě jste zaznamenali, že nepadla zmínka o důležité funkci naskladnění zboží. Ta byla implementována skrze modul identifikace, kterému je věnována následující kapitola.

4.6.5 Identifikace

Mobilní klient byl zamýšlen pro nasazení na mobilní zařízení disponující čtečkou čárových kódů či RFID a proto byla do množiny podporovaných agend zařazena také identifikace. Ta nám umožní jednoduše přečtené zboží naskladnit, případně přímo umístit na doklad faktury, objednávky.

Obrázek 4.17: Formulář pro identifikaci zboží

Formulář je složen z několika tlačítek, jejichž význam si postupně vysvětlíme. Začneme tlačítkem *Čtení z čtečky*. Existence tohoto tlačítka je dána tím, že jsem v průběhu implementace neměl k dispozici zařízení, které by mělo integrovanou čtečku čárových kódu. Z toho důvodu jsem byl nucen implementovat alternativní řešení v podobě simulace čtení kódů. Úkolem tlačítka *Čtení z čtečky* je „přečíst“ čárový kód a „načtené“ zboží umístit do tabulky. V pozadí tohoto „čtení“ je jednoduché generování GUID řetězců, které představují identifikační kód.

Další je dvojice tlačítek *Na objednávku, Na fakturu*. Tyto tlačítka může uživatel využít ve chvíli kdy načte nějaké zboží a chce na něj rovnou vystavit doklad. Stiskem jednotlivých

tlačítek je mu zobrazen formulář s vytvořením nového dokladu faktury či objednávky. Načtené zboží je automaticky převedeno do tabulky položek dokladu. (viz. obrázek 4.15)

V závěru kapitoly pojednávající o skladovém formuláři jsem se zmínil o tom, že funkce naskladnění byla implementována v modulu identifikace. Je tomu tak a celá tato „magie" je ukryta za tlačítkem *Naskladnit*. S touto funkcí také souvisí roletkové menu pro výběr kategorie umístěné nad tabulkou načítaného zboží. Jakmile načteme zboží a vybereme si kategorii, stačí jen stisknout tlačítko *Naskladnit* a zboží je ihned odesíláno na server.

Posledním z množiny tlačítek je *Přidat na doklad*. V tuto chvíli se dostáváme k objasnění funkcionality tlačítka *Přečíst z čtečky* zobrazeného na obrázku 4.15. Pokud uživatel zvolí možnost přidání položek za pomocí čtečky, je mu zobrazen formulář identifikace. V tomto případě jsou všechna ostatní tlačítka, vyjma *Přidat na doklad*, před uživatelem skryta. Načtením a stiskem tohoto tlačítka dojde k převedení zboží na doklad. Jistě jste si všimli, že je přidání zboží na doklad obousměrné. Jedna cesta vede přes identifikaci a druhá přímo z dokladu.

Ještě než ukončíme povídání o identifikaci, zbývá zmínit možnost editace načteného zboží. Každá z položek v tabulce může být editována uživatelem. K dispozici jsou následující údaje – *RFID, Název, Množství, Měrná jednotka*.Editovat lze kterýkoliv z nich, kromě *RFIDu*.

4.7 Offline mód

Mobilní klient pro svůj běh vyžaduje připojení k internetu. O jisté nespolehlivosti připojení jsem se zmínil v úvodních kapitolách. Co se tedy stane, pokud dojde k výpadku? Lze vůbec klienta spustit a používat bez připojení? V této kapitole se budeme věnovat odpovědím na tyto otázky. A začneme rovnou druhou položenou.

Odpověď zní ne, aplikace klienta nelze spustit a používat bez aktivního připojení k internetu. Ačkoliv se to může zdát nečekané, toto rozhodnutí má několik důvodů týkajících se budoucího rozvoje aplikace. Jedním z nich je identifikace uživatele, kde tak jako ve velkém ERP systému, má každý uživatel vlastní účet pod kterým se přihlašuje. Identifikace sehraje důležitou roli, pro kontrolu operací z mobilního zařízení a převzetí zodpovědnosti. Dalším z důvodů je „rozdělení" aplikace na základě načtených práv jednotlivých uživatelů. Jednotlivé části aplikace poté budou přístupné např. jen lidem s daným oprávněním. A další důvody mohou následovat. Pojďme se ale podívat, jak je to se situací při výpadku spojení.

Aplikace mobilního klienta má dva tzv. módy:

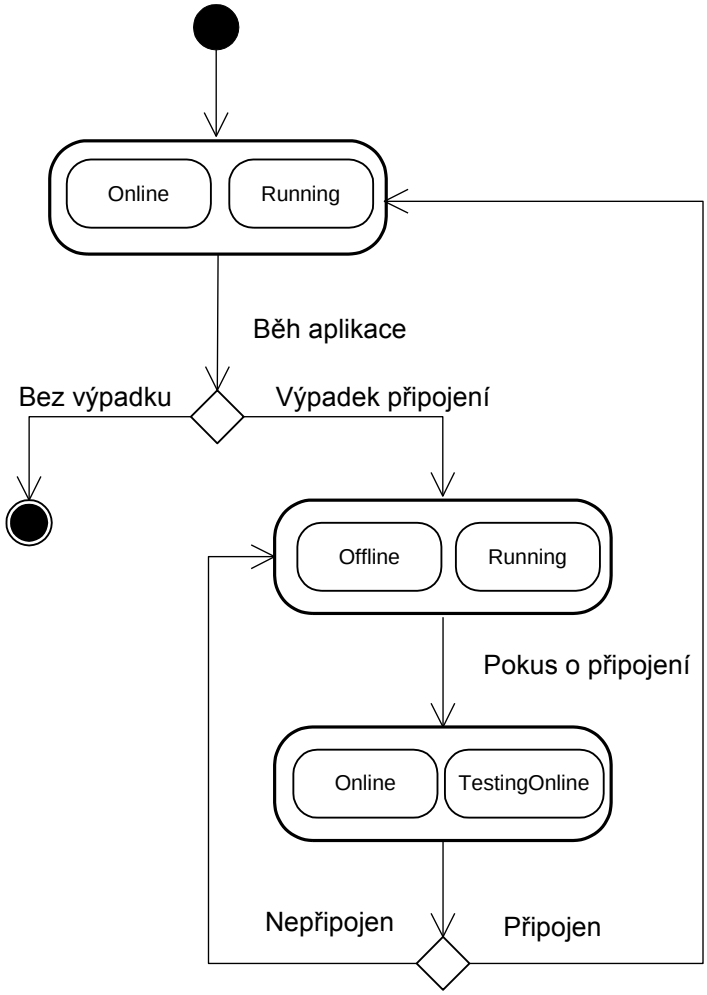
- Online
- Offline

Pokud má uživatel k dispozici připojení k internetu a spustí klienta, za předpokladu že nedojde během nahrávání k výpadku, přejde klient do stavu *Online*. Co to znamená? Znamená to, že klient může odesílat požadavky na server a nemá omezenou funkcionalitu. Opak tvoří mód *Offline* do něhož se aplikace dostane právě po výpadku připojení k internetu. Řízení chování aplikace si vyžádalo zavedení dalších tří pomocných stavů:

- Loading
- Running

- TestingOnline

Stav *Loading* indikuje situaci, kdy aplikace nahrává formuláře, předehrává data ze serveru a připravuje hlavní formulář k jeho zobrazení. Tento proces se děje vždy po spuštění aplikace. *Running* je příznakem toho, že aplikace běží a poslední z uvedených, *TestingOnline*, indikuje pokus o znovu navázání spojení se serverem. Tuto možnost má uživatel díky tlačítku ve formuláři s nastavením klienta. Pokud tedy došlo k výpadku, může se uživatel pokusit znovu navázat spojení.



Obrázek 4.18: Stavový diagram běhu aplikace mobilního klienta

Co není na obrázku 4.18 vyjádřeno, je použití stavu *Loading*. Při inicializaci aplikace, dojde k nastavení stavu *Online* a *Loading* a klient nahrává prostředky potřebné ke své funkci. Během této operace také může dojít k výpadku připojení. Pokud se tak stane, je na obrazovku vypsáno hlášení a klient je ukončen.

Diagram na obrázku 4.18 znázorňuje situaci, kdy došlo k úspěšnému nahrání aplikace. Ta běží a uživatel s ní může pracovat. Je tedy ve stavu *Online* a *Running*. Během práce však

může opět dojít k výpadku připojení k internetu. Klient zareaguje tím, že přejde do stavu *Offline* a pomocný stav *Running* zastává.

Stav *Offline* pro uživatele znamená omezení funkcionality. Veškeré operace vyžadující data ze serveru jsou zamezeny. To se týká prohlížení záznamů a jejich detailů. Tvorba nových entit – firem, dokladů, naskladnění zůstala nezměněna a tyto funkce jsou i v *Offline* stavu aktivní. Toho bylo dosaženo modifikací bázové třídy jednotlivých proxy tříd. Na základě stavu uživatele aplikace rozeznává ukládací operace, a pokud zjistí, že není k dispozici připojení k internetu, ukládá požadavky do stanoveného adresáře.

```
<Action s:mustUnderstand="1"
xmlns="http://schemas.microsoft.com/ws/2005/05/addressing/none">
http://tempuri.org/INpoManager/SaveObject
</Action>
```

Obrázek 4.19: Identifikace akce v hlavičce SOAP zprávy

```
if (elem.ToLower().Contains("save"))
   {
    XmlUtils.SaveMessageToFile(FilenameGenerator.
     GetFilename(this.SaveEntityType, this.SaveFileIndentifier), msg);
   }
```

Obrázek 4.20 : Kód rozeznávající požadavky na uložení entity

Tučně vyznačený název akce z obrázku 4.19 identifikuje službu a metodu, pro kterou je požadavek určen. V těle zprávy se nachází celý objekt objednávky popsaný pomocí XML. Rozeznání odchozího požadavku je založeno na existenci podřetězce „save“ v tomto identifikačním řetězci. Díky mé konvenci v pojmenování služeb, kdy služby ukládající entity obsahují právě zmíněný řetězec, toto řešení funguje. Nicméně v budoucnu by mělo být určitě nahrazeno pokročilejším rozeznáváním. Například z toho důvodu, že názvy metod vystavených službami by se mohly změnit a řešení by přestalo fungovat. Případně zprovoznění by navíc vyžadovalo přepsání kódu.

Ukládání XML zpráv na uložiště mobilního zařízení si vyžádalo vyřešit problém s identifikací a zpětnou konverzí textu do objektu. Protože je entit více druhů, musela být zavedena jejich jednoznačná identifikace. Ta primárně posloužila k rozeznání jejího typu na jehož základě je vybírána proxy třída schopna zasílat požadavky zpět na server. Každá z proxy tříd je potomkem třídy, která se stará o přenos požadavků. Do této třídy byl umístěn následující kód:

```
protected virtual string SaveEntityType
{
     get
       {
          return string.Empty;
       }
}
```

Jedná se o textový řetězec identifikující ukládanou entitu. Slovíčko virtual v hlavičce metody říká, že implementace atributu může být v potomcích třídy „přepsána“. Této vlastnosti bylo využito a jednotlivé proxy třídy definují identifikátor entity, o kterou se třída „stará“. Na obrázku 4.20 je tohoto atributu využito při ukládání entity na uložiště mobilního zařízení. Ukládající metoda využíva ještě jeden atribut – *SaveFileIndentifier*. Tento pomocný řetězec slouží k identifikaci ukládaného souboru a zamezuje vícenasobnému uložení entity. Jeho hodnota je vkládána do názvu a při opětovném uložení stejné entity, dojde k přepsání té původní. Dokud proxy třída neobdrží nový identifikátor, ukladá pořád stejnou entitu.

Vraťme se ještě k obrázku 4.17. Jak jsem již zmínil, existuje formulář s nastavením klienta, na němž se nachází tlačítko, kterým je možné obnovit klienta zpět do stavu *Online* a zpřístupnit tak všechny jeho funkce. Pokud je pokus o obnovení úspěšný a připojení je navázáno, klientův stav se změní na kombinaci *Online, Running*.

Co se ale děje s uloženými entitami v adresáři? Daný adresář je při nahrávání prohlédnut, a pokud jsou nalezeny nějaké uložené záznamy, je uživateli nabídnuta možnost odeslat je na server. Pokud se tak stane, soubory jsou přečteny a z nich „vyrobené“ objekty jsou zasílány na server stejně jako by se jednalo o přímé zasílání požadavků. Tento proces je identický i pro situaci kdy uživatel úspěšně obnoví připojení k internetu.

4.8 Zabezpečení

Aplikace pracující v prostředí internetu, i mimo něj by měly být vždy zabezpečeny. Čím jsou data, se kterými se operuje citlivější, tím větší by mělo být zabezpečení. Ke splnění tohoto úkolu se dnes používají různé techniky. Pro zabezpečení mobilního klienta jsem zvolil technologii certifikátů, o níž si krátce povíme v této kapitole a ukážeme si ukázky nastavení takovéhoto zabezpečení.

Při seznamování s .NET Compact Frameworkem jsme v kapitole 2.4.6 narazili na jakási omezení týkající se samotného zabezpečení aplikací vyvíjených za pomoci WCF. Po prozkoumání všech dostupných možností jsem se rozhodl použít zabezpečení pomocí certifikátů. Jak toto zabezpečení funguje? Na serveru je vygenerován certifikát a v konfiguračním souboru služeb nastaveny parametry pro jeho použití. Tento certifikát je poté nutné distribuovat spolu s klientskou aplikací. Při zaslání požadavku na některou ze služeb je tento certifikát vyžadován. Pokud jej proxy třída neposkytne, server odmítne komunikaci a tím proces končí. Takto ve zkratce probíhá interakce mezi klientem a serverem. Pojďme se podívat jak vypadá nastavení služeb.

První věc, kterou je potřeba udělat je nastavení zabezpečení pro *vazbu* (viz. kapitola 2.4.3), kterou jednotlivé služby využívají. Ačkoliv je v nabídce více šifrovacích algoritmů, jediným podporovaným je *Basic256Rsa15*.

```
<security mode="Message">
 <message clientCredentialType="Certificate"
          algorithmSuite="Basic256Rsa15" />
</security>
```

Druhým krokem je modifikace elementu nesoucí název *chování* (behavior). Každá ze služeb má mimo své *koncové body* (viz. kapitola 2.4.2) i tzv. chování. Toto nastavení zahrnuje různé možnosti, mezi nimiž je i zabezpečení.

```
<serviceCredentials>
 <clientCertificate>
  <certificate
      findValue="xevos"
      storeLocation="LocalMachine"
      storeName="My"
      x509FindType="FindBySubjectName" />
<authentication certificateValidationMode="ChainTrust" />
</clientCertificate>
 <serviceCertificate
      findValue="xevos"
      storeLocation="LocalMachine"
      storeName="My"
      x509FindType="FindBySubjectName" />
</serviceCredentials>
```

Abychom porozuměli tomuto nastavení, popišme si jednotlivé atributy. Z ukázky je jasné, že používáme stejného nastavení certifikátu jak pro klienta, tak pro vlastní službu. Hodnota atributu *findValue* může být libovolná, záleží pouze na tom, kdo certifikát vystavil. Pokud tedy vystavíte vlastnoručně podepsaný certifikát na počítači s názvem xevos, pak nastavení bude vypadat jako v ukázce. Následujícím atributem je *storeLocation*. Jedná se výčtový typ, nabývající hodnot *CurrentUser* a *LocalMachine* definující, kde se bude certifikát hledán. Další atribut je taktéž výčtový, jeho hodnoty můžete nalézt zde [10]. Pro nás je podstatné, že specifikuje adresář v umístění *storeLocation.* Jeho hodnota "My" je v počeštěných operačních systémech přeložena jako „Osobní". Posledním atributem je *x509FindType* jehož definicí říkáte jakým způsobem má hledat certifikát se zadaným řetězcem *findValue*. Může nabývat mnoha hodnot a pro detailní studium doporučuji [11].

Element autentizace má v tomto případě jediný atribut. Tím je *certificateValidationMode*, určující typ validace certifikátu. Je to výčtový typ podporující hodnoty *PeerTrust , ChainTrust , ChainOrPeerTrust,Custom, None.* .NET Compact Framework ovšem podporuje jen hodnoty *ChainTrust* a *None*. Námi zvolená hodnota říká, že klientský certifikát bude validován oproti certifikátu uloženému v kořenovém adresáři serveru.

Zbývají poslední dva kroky k úspěšnému zabezpečení pomocí certifikátu. Prvním z nich je nasazení modifikovaného konfiguračního souboru na server. Následuje klientské vygenerovaní proxy tříd s novým nastavením. (viz. kapitola 2.4.5) Druhým krokem je instalace certifikátu na mobilní zařízení a lehká modifikace kódu proxy tříd jako v následující ukázce.

```
NpoManagerClient proxy = new NpoManagerClient();

proxy.ClientCredentials.ClientCertificate.
SetCertificate(StoreLocation.CurrentUser,
               StoreName.My,
               X509FindType.FindBySubjectName,
               "xevos");
proxy.ClientCredentials.ServiceCertificate.
```

```
SetDefaultCertificate(StoreLocation.CurrentUser,
                      StoreName.My,
                      X509FindType.FindBySubjectName,
                      "xevos");
```

Z kódu je zřejmé, že musí dojít ke stejnému nastavení jako na serveru. Pokud by se kód lišil, nebylo by možné zasílat požadavky na služby a server by komunikaci odmítl.

Kapitola 5
Závěr

Cílem práce byla implementace aplikace pracující nad větším ERP systémem. Díky tomu, že se jednalo o aplikaci pro mobilní platformu, jsem se musel seznámit hned s několika technologiemi.

Práce probíhala na obou stranách „barikády“, tedy jak na serveru, tak na klientovi. Různorodost implementace vedla k tomu, že se práce nestala stereotypní. Různé postupy na obou stranách vedly k prohloubení mých znalostí a zejména získání důležitých zkušeností. Jmenujme např. návrhové vzory či vícevrstvou architekturu.

Ačkoliv zatím neproběhly testy v reálném provozu, klient byl otestován na několika mobilních zařízeních. Na jednom starším zařízení vše fungovalo podle zadání a nevyskytly se téměř žádné problémy. Ovšem na novějším zařízení byla situace jiná. Tento novější model mobilního zařízení disponoval velkým rozlišením, na které není aplikace připravena. Důsledkem bylo zmenšení celého uživatelského rozhraní.

V obou případech byla zařízení připojena do internetu pomocí bezdrátového připojení.(Wi-Fi) Klient na tomto připojení vykazoval přijatelnou odezvu.

Samotná aplikace bude jistě časem upravena a obohacena o novou funkcionalitu, tak aby mohla být v budoucnu nasazena v produkčním prostředí. Mezi vylepšení by se určitě dalo zařadit přidání systému rolí a oprávnění, rozšíření o nové agendy a propracovanější podpora pro práci bez internetového připojení.

Nejvíce si cením získaných zkušeností z oblasti vývoje pro mobilní platformu. Samozřejmě nesmím zapomenout na dnes široce využívanou technologii WCF, díky níž jsem poznal, jak snadná je tvorba aplikací typu klient-server.

Literatura

[1] Microsoft. *Msdn.microsoft.com* [online]. 2007 [cit. 2010-04-12]. Windows Communcation Foundation Bindings. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/ms733027.aspx>

[2] *W3.org* [online]. 2007 [cit. 2010-04-12]. SOAP Version 1.2 Part 0: Primer (Second Edition). Dostupné z WWW: <http://www.w3.org/TR/soap12-part0/>

[3] *W3.org* [online]. 2009 [cit. 2010-04-12]. SOAP over Java Message Service 1.0. Dostupné z WWW: <http://www.w3.org/TR/2009/CR-soapjms-20090604/>

[4] *Mono-project.com* [online]. 2004 [cit. 2010-04-12]. Mono. Dostupné z WWW: <http://www.mono-project.com/Main_Page>

[5] ARNOTT, Andrew. *Blogs.msdn.com* [online]. 2007-08-21 [cit. 2010-04-12]. The WCF subset supported by NetCF. Dostupné z WWW: <http://blogs.msdn.com/andrewarnottms/archive/2007/08/21/the-wcf-subset-supported-by-netcf.aspx>

[6] Microsoft. *Msdn.microsoft.com* [online]. 2010 [cit. 2010-04-12]. Messaging in the .NET Compact Framework. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/bb397842.aspx>

[7] Microsoft. *Msdn.microsoft.com* [online]. 2009 [cit. 2010-04-12]. Types of Concurrency Control. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/ms189132(v=SQL.100).aspx>

[8] Microsoft. *Msdn.microsoft.com* [online]. 2010 [cit. 2010-04-12]. Implementing Singleton in C#. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/ms998558.aspx>

[9] PURDY, Doug; RICHTER, Jeffrey. *Msdn.microsoft.com* [online]. 2002 [cit. 2010-04-12]. Exploring the Observer Design Pattern. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/Ee817669(pandp.10).aspx>

[10] Microsoft. *Msdn.microsoft.com* [online]. 2010 [cit. 2010-04-12]. StoreName Enumeration. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/system.security.cryptography.x509certificates.storename.aspx>

[11] Microsoft. *Msdn.microsoft.com* [online]. 2010 [cit. 2010-04-12]. X509FindType Enumeration. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/system.security.cryptography.x509certificates.x509findtype.aspx>

[12] Microsoft. *Msdn.microsoft.com* [online]. 2010 [cit. 2010-04-12]. Model-View-Controller. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/ms978748.aspx>

[13] Microsoft. *Msdn.microsoft.com* [online]. 2010 [cit. 2010-04-12]. DataTable Class. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/system.data.datatable.aspx>

Příloha A – Obsah přiloženého CD

Na přiloženém CD lze nalézt kompletní aplikaci mobilního klienta, včetně knihovny a nastavení služeb pro serverovou část řešení. CD má následující strukturu:

server	adresář obsahující zdrojové kódy pro serverovou část řešení
client	adresář obsahující zdrojové kódy mobilního klienta
text	zde je uložen tento text ve formátech PDF a DOC
doc	adresář obsahující dokumentaci API ke klientské části řešení